# Operating System/2™ Programming Tools and Information Version 1.2

## I/O Subsystems and Device Support

## Volume 1 Device Drivers

64F3959

Operating System/2™
Programming Tools
and Information
Version 1.2

I/O Subsystems and Device Support

Volume 1
Device Drivers

**First Edition (September 1989)**

**The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

© Copyright International Business Machines Corporation 1986, 1987, 1988

Note to US Government users — Documentation related to Restricted Rights — Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

# Special Notices

The following names, used in this publication, are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries:

IBM
C/2
Macro Assembler/2
Operating System/2
OS/2
Presentation Manager
Systems Application Architecture.

The following names, used in this publication, are trademarks or registered trademarks of other companies:

| | |
|---|---|
| CodeView | Microsoft Corporation |
| PostScript | Adobe Systems Incorporated. |

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license enquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Armonk, NY 10504.

# Preface

The Operating System/2 Version 1.2 (OS/2) *I/O Subsystems and Device Support Volume 1: Device Drivers* contains device driver concepts for system programmers as well as application programmers. Programmers can use this book to write their own device drivers and subsystems.

*I/O Subsystems and Device Support Volume 2: Presentation Drivers* contains presentation driver concepts and interfaces. The programmer can use these device interfaces to write their own presentation drivers.

The Programming Tools and Information library is intended for professional application developers knowledgeable in at least one programming language in which OS/2 programs can be written. The information in the Programming Tools and Information library assumes you are new to programming with OS/2 and the Presentation Manager. You should understand the OS/2 services available to users. Further information on these can be obtained from the OS/2 Product library.

The Operating System/2 Version 1.2 *Programming Overview* introduces the programming concepts that you should understand before you begin developing applications to run on OS/2, and describes the set of books, tools, programming aids, and sample programs that make up the OS/2 Programming Tools and Information Library.

Details of publications for OS/2 Version 1.2 and related products are shown in the Library Diagram on page vi.

In this book:

**Chapter 1 - Introduction**
This chapter provides a general description of OS/2 device drivers and presentation drivers, how they fit into the operating system, how they are are different.

**Chapter 2 - Device Driver Overview**
This chapter provides a detailed description of OS/2 device driver installation, I/O support for applications, initialization, and architecture.

**Chapter 3 - Designing an OS/2 Device Driver**
This chapter provides a detailed description of device driver design and build considerations.

**Chapter 4 - Device Driver Strategy Commands**
This chapter provides a detailed description of the device driver strategy command entry points that a device driver must support.

**Chapter 5 - Device Helper Services**
This chapter provides a detailed description of the device helper services that are available to all device drivers.

**Chapter 6 - Device Drivers**
This chapter provides a detailed description of OS/2 device drivers - the ASYNC, Mouse and Pointer Draw, Clock, Console (Screen and Keyboard), Screen, Keyboard, Disk, Video, and Printer device drivers.

**Chapter 7 - Generic IOCTL commands**
This chapter describes the IOCTL interfaces supported by the various OS/2 device drivers.

**Chapter 8 - Character Device Monitors**
This chapter provides a detailed description of the OS/2 Monitor Mechanism, including the monitor application and monitor support in device drivers.

**Chapter 9 - I/O Privilege Level Code Segments (IOPL)**
This chapter provides a description of IOPL and guidelines for using IOPL code segments.

# The OS/2 1.2 Library and Related Products

### OS/2 Product

IBM Operating System/2
Standard Edition
Version 1.2

Getting Started
Using Advanced Features
Product Information

6024926 3.5″ diskette
6024930 5.25″ diskette

### OS/2 Programming Tools and Information

IBM Operating System/2
Version 1.2

Installation
Programming Overview
Programming Guide
Building Programs
Dialog Tag Language Guide and Reference
Dialog Manager Guide and Reference
Dialog Manager and Dialog Tag Language Reference Summary
Programming Reference (3 books)
Bindings Reference for Presentation Manager (4 books for COBOL/2, FORTRAN/2, C/2, and Macro Assembler/2)
I/O Subsystems and Device Support (2 books)
Systems Application Architecture
Common User Access: Advanced Interface Design Guide

6024929

### Separate Order (no charge)

Keyboards and Code Pages

6280345

### Available Separately

Command Reference

6024928

Service Coordinator's Guide

15F2214

### Programming Languages

IBM Basic Compiler/2
Version 1.0 6280179

IBM Macro Assembler/2
Version 1.0 6280181

IBM Pascal Compiler/2
Version 1.0 6280183

IBM FORTRAN/2
Version 1.02 6280185

IBM COBOL/2
Version 1.0 6280207

IBM C/2
Version 1.1 6280284

### Systems Application Architecture

An Overview

GC26-4341

Writing Applications: A Design Guide

SC26-4362

Common User Access: Advanced Interface Design Guide
SC26-4582

Common User Access: Basic Interface Design Guide
SC26-4583

### Common Programming Interface

C Reference - Level 2

SC09-1308

COBOL Reference

SC26-4354

Dialog Reference

SC26-4356

FORTRAN Reference

SC26-4357

Procedures Language Reference
SC26-4358

Presentation Reference

SC26-4359

# Contents


**Chapter 1. Introduction** . . . . . 1-1
OS/2 Device Drivers . . . . . 1-3
386 Programming Considerations . . . . . 1-4
OS/2 Presentation Drivers . . . . . 1-4

**Chapter 2. Device Driver Overview** . . . . . 2-1
Types of Device Drivers . . . . . 2-1
Installation of External Loadable Device Drivers . . . . . 2-1
Device Driver Profile . . . . . 2-2
:CONFIG . . . . . 2-2
:FILES . . . . . 2-2
Sample Device Driver Profile . . . . . 2-2
Application I/O to Devices . . . . . 2-3
I/O Support for the OS/2 Mode . . . . . 2-5
I/O Support For The DOS Mode . . . . . 2-6
Device Driver Initialization . . . . . 2-7
Device Driver INIT-Time Function Call Summary . . . . . 2-8
Replacing Character Device Drivers . . . . . 2-8
Compatibility with Previous-Level Device Drivers . . . . . 2-9
Initialization of Previous-Level Device Drivers . . . . . 2-9
Device Driver Architecture . . . . . 2-10
Device Driver Program Model . . . . . 2-10
Device Driver Header . . . . . 2-11
Device Driver Components . . . . . 2-13
OS/2 Device Driver Contexts . . . . . 2-14
OS/2 Device Driver Operations . . . . . 2-15
Request Packet Queue Management . . . . . 2-16
Memory Management . . . . . 2-16
Semaphore Management . . . . . 2-17
Character Queue Management . . . . . 2-17
Hardware Interrupt Management . . . . . 2-18

**Chapter 3. Designing an OS/2 Device Driver** . . . . . 3-1
Design Considerations . . . . . 3-1
Designing for Performance . . . . . 3-1
Requesting OS/2 Services . . . . . 3-1
Limiting the Number of Nested Interrupts . . . . . 3-2
Inter-Device Driver Communication . . . . . 3-2
Device Monitor Support in Character Device Drivers . . . . . 3-3
Device Support . . . . . 3-4
DOS Mode Generic IOCtl Support . . . . . 3-5
DOS Mode Software Interrupt Support . . . . . 3-5
Using Advanced BIOS . . . . . 3-6
Device Driver Data Segment . . . . . 3-6
Obtaining a Logical ID . . . . . 3-7
Calling Advanced BIOS Services . . . . . 3-7
Mapping Device Names to LID . . . . . 3-7
Handling ABIOS Requests . . . . . 3-8
Writing a Device Driver using Advanced BIOS . . . . . 3-8
Using DevHlp Services for Address Conversion . . . . . 3-11
Building a Device Driver . . . . . 3-13
Device Driver Sample Program (DEMODD.ASM) . . . . . 3-14

**Chapter 4. Device Driver Strategy Commands** . . . . . 4-1
Request Packets . . . . . 4-1
Summary of Device Driver Commands . . . . . 4-4
0H / INIT Initialize Device Driver . . . . . 4-5
1H / MEDIA CHECK Check the Media . . . . . 4-7
2H / BUILD BPB Build BIOS Parameter Block . . . . . 4-9

Boot Sector Format . . . . . 4-10
4H, 8H, 9H / READ or WRITE Perform I/O To A Device . . . . . 4-11
5H / NONDESTRUCTIVE READ NO WAIT Nondestructive Input . . . . . 4-12
6H, AH / STATUS Input or Output Status . . . . . 4-13
7H, BH / FLUSH Input or Output Flush . . . . . 4-14
DH, EH / OPEN or CLOSE Open / Close Device . . . . . 4-15
FH / REMOVABLE MEDIA Check for Removable Media . . . . . 4-16
10H / GENERIC IOCtl I/O Control for Devices . . . . . 4-17
11H / RESET MEDIA Reset Uncertain Media Condition . . . . . 4-19
12H, 13H / LOGICAL DRIVE Get/Set Logical Drive Mapping . . . . . 4-20
14H / DEINSTALL Terminate the Device Driver . . . . . 4-21
DEINSTALL Considerations . . . . . 4-21
16H / PARTITIONABLE FIXED DISKS General query of device support . . . . . 4-22
17H / GET FIXED DISK/LOGICAL UNIT MAP . . . . . 4-23

**Chapter 5. Device Helper Services** . . . . . 5-1
DevHlp Services and Function Codes . . . . . 5-1
DevHlp Services and Device Contexts . . . . . 5-2
DevHlp Interfaces . . . . . 5-6
ABIOSCall Invoke ABIOS function . . . . . 5-7
ABIOSCommonEntry Invoke ABIOS Common Entry Point . . . . . 5-8
AllocGDTSelector Allocate GDT Selectors . . . . . 5-9
AllocPhys Allocate Fixed Block of Physical Memory . . . . . 5-10
AllocReqPacket Get a Request Packet . . . . . 5-11
AttachDD Attach to a Device Driver . . . . . 5-12
Block This Thread From Running . . . . . 5-13
DeRegister Remove Monitor . . . . . 5-15
DevDone Flag I/O Complete . . . . . 5-16
EOI Issue an End-Of-Interrupt . . . . . 5-17
FreeLIDEntry Release a Logical ID . . . . . 5-18
FreePhys Free Physical Memory . . . . . 5-19
FreeReqPacket Free an Allocated Request Packet . . . . . 5-20
GetDOSVar This routine is used to return the address of a kernel variable. . . . . 5-21
GetLIDEntry Get a Logical ID . . . . . 5-22
InternalError Signal an Internal Error . . . . . 5-23
Lock Lock Memory Segment . . . . . 5-24
MonFlush Flush Data from Monitor Chain . . . . . 5-26
MonitorCreate Create a Monitor Chain . . . . . 5-27
MonWrite Give Data to Monitors . . . . . 5-29
PhysToGDTSelector Map Physical Address to a GDT Selector . . . . . 5-30
PhysToUVirt Map Physical To User Virtual Address . . . . . 5-31
PhysToVirt Map Physical Address to Virtual Address . . . . . 5-33
ProtToReal Change Mode from Protect to Real Mode . . . . . 5-35
PullParticular Remove Specific Request From List . . . . . 5-37
PullReqPacket Remove Request From List . . . . . 5-38
PushReqPacket Add Request To List . . . . . 5-39
QueueFlush Clear Character Queue . . . . . 5-40
QueueInit Initialize Character Queue . . . . . 5-41
QueueRead Read a Character From a Queue . . . . . 5-42
QueueWrite Put Character into Queue . . . . . 5-43
RealToProt Change Mode from Real to Protect Mode . . . . . 5-44
Register Add Monitor . . . . . 5-46
RegisterStackUsage Indicate Stack Usage . . . . . 5-47
ResetTimer Reset Timer Handler . . . . . 5-48
ROMCritSection Flag Critical Section of Execution . . . . . 5-49
Run Release Blocked Thread . . . . . 5-50
SchedClockAddr Get system clock routine . . . . . 5-51
SemClear Release a Semaphore . . . . . 5-53
SemHandle Obtain a Semaphore Handle . . . . . 5-54
SemRequest Claim a Semaphore . . . . . 5-56
SendEvent Indicate an Event . . . . . 5-57
SetIRQ Set Hardware Interrupt Handler . . . . . 5-58
SetROMVector Set DOS Mode Software Interrupt Vector . . . . . 5-59

SetTimer Set Timer Handler . . . . . 5-60
SortReqPacket Insert Request In Sorted Order To List . . . . . 5-61
TCYield Yield the CPU . . . . . 5-62
TickCount Modify timer . . . . . 5-63
Unlock Unlock Memory Segment . . . . . 5-64
UnPhysToVirt Mark Completion of Virtual Address Use . . . . . 5-65
UnSetIRQ Remove Hardware Interrupt Handler . . . . . 5-66
VerifyAccess Verify Access to Memory . . . . . 5-67
VirtToPhys Map Virtual Address to Physical Address . . . . . 5-68
Yield Relinquish the CPU . . . . . 5-69

**Chapter 6. Device Drivers** . . . . . 6-1
ASYNC (RS232-C) Communications Device Driver . . . . . 6-1
Hardware Support . . . . . 6-1
IBM Personal Computer AT Adapter Support . . . . . 6-2
IBM PS/2 Adapter Support . . . . . 6-2
Attachment Support . . . . . 6-2
RS232-C Interface . . . . . 6-3
Hardware Support for Extended Hardware Buffering . . . . . 6-3
INS 8250, INS 8250-B Considerations . . . . . 6-3
Supported Bit (Baud) Rates on 16450 and Compatibles . . . . . 6-4
ASYNC (RS232-C) Device Driver Features . . . . . 6-4
Output Modem Control Signals . . . . . 6-6
Input Modem Control Signals . . . . . 6-6
Logical Flow Control (XON/XOFF) . . . . . 6-7
Support for Extended Hardware Buffering . . . . . 6-7
Line Characteristics . . . . . 6-7
Break and Error Processing . . . . . 6-7
State of the COM Port . . . . . 6-7
Event Notification . . . . . 6-8
Error Alert Generation . . . . . 6-8
States of the ASYNC Device Driver . . . . . 6-8
Baud Rate . . . . . 6-9
Data Bits . . . . . 6-9
Parity . . . . . 6-9
Stop Bits . . . . . 6-9
DTR and RTS . . . . . 6-10
DTR Control Mode . . . . . 6-10
RTS Control Mode . . . . . 6-10
Transmitting Break . . . . . 6-11
COM Event Word and COM Error Word . . . . . 6-11
Output Handshaking using CTS, DSR, DCD . . . . . 6-11
Input Sensitivity Using DSR . . . . . 6-11
Automatic Transmit Flow Control (XON/XOFF) . . . . . 6-12
Automatic Receive Flow Control (XON/XOFF) . . . . . 6-12
XON/XOFF characters . . . . . 6-13
Error Replacement Character Processing . . . . . 6-13
Error Replacement Character . . . . . 6-13
Break Replacement Character Processing . . . . . 6-13
Break Replacement Character . . . . . 6-13
Null Stripping . . . . . 6-13
Write Time-out State . . . . . 6-14
Write Time-out Value . . . . . 6-14
Read Time-out State . . . . . 6-14
Read Time-out Value . . . . . 6-14
Transmit Immediate . . . . . 6-14
Extended Hardware Buffering . . . . . 6-15
Reserved Device Names- COM1-N . . . . . 6-16
IBM Personal Computer AT Considerations- COM1, COM2 . . . . . 6-16
IBM PS/2 Considerations- COM1 - 3 . . . . . 6-16
Initialization / Resource Management . . . . . 6-16
IBM Personal Computer AT Initialization Considerations . . . . . 6-17
IBM PS/2 Initialization Considerations . . . . . 6-18

Data Translation / Monitor Support /Spooler Support . . . . . 6-18
ASYNC Communication Device Driver Interfaces . . . . . 6-18
File System Requests . . . . . 6-19
Access Authorization . . . . . 6-21
ASYNC (RS232-C) Generic IOCtl Command Summary . . . . . 6-22
DOS Mode Considerations / Restrictions . . . . . 6-23
Control Panel Condiderations . . . . . 6-23
Performance . . . . . 6-23
Performance in Character (NS-16450) Mode . . . . . 6-24
Enabling Extended Hardware Buffering . . . . . 6-24
Pointer Draw (Screen) Device Driver . . . . . 6-26
Full Draw Support . . . . . 6-26
Disabled Draw Support . . . . . 6-26
DOS Mode Support . . . . . 6-26
Mouse Pointer Draw Implementation . . . . . 6-27
Pointer Draw IDC Interface Data Structures . . . . . 6-30
Mouse Session Control Block (MSCB) . . . . . 6-33
Mouse Device Driver . . . . . 6-37
Mouse Device Overview . . . . . 6-37
Mouse Inter Device Driver (IDC) Interfaces . . . . . 6-39
OS/2 Mode Mouse Support . . . . . 6-44
OS/2 Mode Mouse API . . . . . 6-46
Mouse Monitors . . . . . 6-48
DOS Mode Mouse Support . . . . . 6-49
DOS Mode INT 33H Mouse API . . . . . 6-50
INT 33H-0 Mouse Installed Flag and Reset . . . . . 6-52
INT 33H-1 Show Mouse Pointer . . . . . 6-52
INT 33H-2 Hide Mouse Pointer . . . . . 6-53
INT 33H-3 Get Mouse Pointer Position & Button Status . . . . . 6-53
INT 33H-4 Set Mouse Pointer Position . . . . . 6-54
INT 33H-5 Get Button Press Information . . . . . 6-54
INT 33H-6 Get Button Release Information . . . . . 6-55
INT 33H-7 Set Min & Max Horiz Position . . . . . 6-56
INT 33H-8 Set Min & Max Vertical Position . . . . . 6-56
INT 33H-9 Set Graphics Pointer Shape . . . . . 6-57
INT 33H-10 Set Text Pointer Shape . . . . . 6-57
INT 33H-11 Read Mouse Motion Counters . . . . . 6-58
INT 33H-12 Set User-Defined Subroutine Input Mask . . . . . 6-58
INT 33H-13 Light Pen Emulation Mode ON . . . . . 6-59
INT 33H-14 Light Pen Emulation Mode OFF . . . . . 6-59
INT 33H-15 Set Mickey/Pixel Ratio . . . . . 6-59
INT 33H-16 Conditional Off . . . . . 6-60
INT 33H-19 Set Double Speed Threshold . . . . . 6-60
INT 33H-20 Swap User-Defined Subroutine . . . . . 6-61
INT 33H-21 Query Save Mouse State Storage Requirements . . . . . 6-61
INT 33H-22 Save Mouse Driver State . . . . . 6-62
INT 33H-23 Restore Mouse Driver State . . . . . 6-62
CLOCK$ Device Driver . . . . . 6-63
Console Device Drivers (Screen and Keyboard) . . . . . 6-64
Keyboard Device Driver KBD$ . . . . . 6-64
Keyboard Initialization . . . . . 6-65
Keyboard Run Time Operation . . . . . 6-65
Keystroke Monitors . . . . . 6-66
Special Key Processing . . . . . 6-70
Compatibility Operations . . . . . 6-71
EGA.SYS and INT 2F Screen Switch Notification . . . . . 6-72
EGA.SYS Device Driver . . . . . 6-72
EGA Register Interface . . . . . 6-72
Function F0 - Read One Register . . . . . 6-74
Function F1 - Write One Register . . . . . 6-75
Function F2 - Read Register Range . . . . . 6-76
Function F3 - Write Register Range . . . . . 6-76
Function F4 - Read Register Set . . . . . 6-77

Function F5 - Write Register Set . . . 6-78
Function F6 - Revert to Default Registers . . . 6-79
Function F7 - Define Default Register Table . . . 6-79
Function F8 - Read Default Register Table . . . 6-80
Function FA - Interrogate Driver . . . 6-81
INT 2F Screen Switch Notification . . . 6-82
Using Extended Screen and Keyboard Control (ANSI.SYS, ANSICALL.DLL) . . . 6-83
Limitations/Restrictions . . . 6-83
Control Sequence Syntax . . . 6-83
Cursor Control Sequences . . . 6-84
Restore Cursor Position . . . 6-85
Erasing . . . 6-85
Controlling Display Mode . . . 6-86
Keyboard Key Reassignment . . . 6-87
Fixed Disk and Diskette Device Driver . . . 6-89
Configuration . . . 6-89
Performance Considerations . . . 6-89
Software Disk Cache . . . 6-90
Interrupt 13H Support . . . 6-90
EXTDSKDD.SYS Support . . . 6-90
DOS Extended Volume Architecture . . . 6-90
Installing Block Devices in the Extended Partition . . . 6-92
Creating Block Devices in the Extended DOS Partition . . . 6-92
Deleting Block Devices in the Extended DOS Partition . . . 6-93
BPB and Get Device Parameters for Extended Volumes . . . 6-95
Category 8 Generic IOCtl Commands . . . 6-95
Category 9 Generic IOCtl Commands . . . 6-96
Type 6 Partition . . . 6-96
Type 7 Partition . . . 6-98
Video Device Drivers . . . 6-99
Base Video Handlers . . . 6-99
Video Device Handler Identification . . . 6-99
Video Device Chaining . . . 6-100
Primary Display Identification . . . 6-100
WrtToScrn/Panic Write Support . . . 6-101
Running System Installation . . . 6-101
Using Loadable Device Drivers . . . 6-101
Video Device Handler Interfaces . . . 6-101
DLL Initialization . . . 6-102
DevEnable . . . 6-102
InitEnable . . . 6-103
Text Buffer Update . . . 6-104
Initialize Environment . . . 6-107
Save Environment . . . 6-107
Restore Environment . . . 6-109
Return Config Info . . . 6-109
Get DBCS Display Info . . . 6-110
Get Color Lookup Table . . . 6-110
Set Color Lookup Table . . . 6-111
Get Cursor Info . . . 6-112
Set Cursor Info . . . 6-113
Get Font . . . 6-114
Set Font . . . 6-115
Get Mode . . . 6-115
Set Mode . . . 6-116
Get Palette Registers . . . 6-116
Set Palette Registers . . . 6-117
Get Phys Buf . . . 6-117
Free Phys Buf . . . 6-118
Get Variable Info . . . 6-118
Set Variable Info . . . 6-120
Terminate Environment . . . 6-122
Print Screen . . . 6-122

Write TTY . . . . . 6-123
Get LVB Info . . . . . 6-125
Level 0 Device Driver Interfaces . . . . . 6-125
INIT ( 00H ) - Initialize the Device . . . . . 6-126
OPEN ( 0DH ) - Open the Device . . . . . 6-126
GENERIC IOCtl ( 10H ) - Send I/O Requests to the Device . . . . . 6-126
DD Initialization . . . . . 6-126
The Printer Device Driver . . . . . 6-128
Overview . . . . . 6-128
Replacing the Printer Device Driver . . . . . 6-129
Code Page Support . . . . . 6-129
Printer Monitors . . . . . 6-129
IOCTL support of Code Page and Font Switching . . . . . 6-131
Timeout Processing . . . . . 6-132
Printer Device Driver Interfaces . . . . . 6-132
Request Packet Interface . . . . . 6-133
Generic IOCTL Interface . . . . . 6-137
Interrupt 17H Interface . . . . . 6-138
Hardware Interrupt (8259) Interface . . . . . 6-140
Monitor Dispatcher Interface . . . . . 6-140

**Chapter 7. Generic IOCtl Commands** . . . . . 7-1
Category Code . . . . . 7-1
Function Code . . . . . 7-1
Generic IOCtl Example . . . . . 7-1
Category 1 ASYNC (RS232-C) Control IOCtl Commands . . . . . 7-5
Category 1 – Function 41H . . . . . 7-6
Category 1 – Function 42H . . . . . 7-7
Category 1 – Function 44H . . . . . 7-9
Category 1 – Function 45H . . . . . 7-10
Category 1 – Function 46H . . . . . 7-11
Category 1 – Function 47H . . . . . 7-13
Category 1 – Function 48H . . . . . 7-14
Category 1 – Function 4BH . . . . . 7-15
Category 1 – Function 53H . . . . . 7-16
Category 1 – Function 61H . . . . . 7-32
Category 1 – Function 62H . . . . . 7-33
Category 1 – Function 64H . . . . . 7-34
Category 1 – Function 65H . . . . . 7-36
Category 1 – Function 66H . . . . . 7-37
Category 1 – Function 67H . . . . . 7-38
Category 1 – Function 68H . . . . . 7-39
Category 1 – Function 69H . . . . . 7-40
Category 1 – Function 6DH . . . . . 7-41
Category 1 – Function 72H . . . . . 7-42
Category 1 – Function 73H . . . . . 7-43
Category 3 Pointer Draw Control IOCtl Commands . . . . . 7-46
Category 3 – Function 72H . . . . . 7-47
Category 3 Video Control IOCtl Commands . . . . . 7-48
Category 3 – Function 73H . . . . . 7-49
Category 3 Screen Control IOCtl Commands . . . . . 7-50
Category 3 – Function 70H . . . . . 7-51
Category 3 – Function 71H . . . . . 7-52
Category 3 – Function 74H . . . . . 7-53
Category 3 – Function 75H . . . . . 7-54
Category 4 Keyboard Control IOCtl Commands . . . . . 7-55
Category 4 – Function 50H . . . . . 7-56
Notes about the Code Page . . . . . 7-57
Category 4 – Function 51H . . . . . 7-66
Category 4 – Function 52H . . . . . 7-67
Category 4 – Function 53H . . . . . 7-68
Category 4 – Function 54H . . . . . 7-69
Category 4 – Function 55H . . . . . 7-70

Category 4 – Function 56H . . . . . 7-71
Category 4 – Function 57H . . . . . 7-73
Category 4 – Function 58H . . . . . 7-74
Category 4 – Function 59H . . . . . 7-75
Category 4 – Function 5AH . . . . . 7-76
Category 4 – Function 5CH . . . . . 7-77
Category 4 – Function 5DH . . . . . 7-78
Category 4 – Function 5EH . . . . . 7-79
Category 4 – Function 71H . . . . . 7-80
Category 4 – Function 72H . . . . . 7-81
Category 4 – Function 73H . . . . . 7-82
Category 4 – Function 74H . . . . . 7-83
Category 4 – Function 75H . . . . . 7-84
Category 4 – Function 76H . . . . . 7-85
Category 4 – Function 77H . . . . . 7-86
Category 4 – Function 78H . . . . . 7-87
Category 4 – Function 79H . . . . . 7-88
Category 4 – Function 7AH . . . . . 7-90
Category 4 – Function 7BH . . . . . 7-91
Category 5 Printer Control IOCtl Commands . . . . . 7-92
Category 5 – Function 42H . . . . . 7-93
Category 5 – Function 44H . . . . . 7-94
Category 5 – Function 46H . . . . . 7-95
Category 5 – Function 48H . . . . . 7-96
Category 5 – Function 62H . . . . . 7-98
Category 5 – Function 64H . . . . . 7-99
Category 5 – Function 66H . . . . . 7-100
Category 5 – Function 69H . . . . . 7-101
Category 5 – Function 6AH . . . . . 7-102
Category 7 Mouse Control IOCtl Commands . . . . . 7-103
Category 7 – Function 50H . . . . . 7-104
Category 7 – Function 51H . . . . . 7-105
Category 7 – Function 52H . . . . . 7-109
Category 7 – Function 53H . . . . . 7-110
Category 7 – Function 54H . . . . . 7-111
Category 7 – Function 56H . . . . . 7-112
Category 7 – Function 57H . . . . . 7-114
Category 7 – Function 58H . . . . . 7-115
Category 7 – Function 59H . . . . . 7-116
Category 7 – Function 5AH . . . . . 7-117
Category 7 – Function 5BH . . . . . 7-119
Category 7 – Function 5CH . . . . . 7-122
Category 7 – Function 5DH . . . . . 7-123
Category 7 – Function 60H . . . . . 7-124
Category 7 – Function 61H . . . . . 7-125
Category 7 – Function 62H . . . . . 7-126
Category 7 – Function 63H . . . . . 7-127
Category 7 – Function 64H . . . . . 7-128
Category 7 – Function 65H . . . . . 7-129
Category 7 – Function 66H . . . . . 7-130
Category 7 – Function 67H . . . . . 7-131
Category 7 – Function 68H . . . . . 7-132
Category 7 – Function 6AH . . . . . 7-133
Category 8 Logical Disk Control IOCtl Commands . . . . . 7-134
Category 8 – Function 00H . . . . . 7-135
Category 8 – Function 01H . . . . . 7-136
Category 8 – Function 02H . . . . . 7-137
Category 8 – Function 03H . . . . . 7-138
Category 8 – Function 04H . . . . . 7-139
Category 8 – Function 20H . . . . . 7-140
Category 8 – Function 21H . . . . . 7-141
Category 8 – Function 43H . . . . . 7-142
Category 8 – Functions 44H, 64H, 65H . . . . . 7-144

Category 8 – Function 45H . . . . . 7-146
Category 8 – Function 63H . . . . . 7-148
Category 9 Physical Disk Control IOCtl Commands . . . . . 7-150
Category 9 – Function 00H . . . . . 7-151
Category 9 – Function 01H . . . . . 7-152
Category 9 – Functions 44H, 64H, 65H . . . . . 7-153
Category 9 – Function 63H . . . . . 7-155
Category 10 Character Device Monitor IOCtl Commands . . . . . 7-156
Category 10 – Function 40H . . . . . 7-157
Category 11 General Device Control IOCtl Commands . . . . . 7-159
Category 11 – Function 01H . . . . . 7-160
Category 11 – Function 02H . . . . . 7-161
Category 11 – Function 41H . . . . . 7-162
Category 11 – Function 60H . . . . . 7-163

**Chapter 8. Character Device Monitors** . . . . . 8-1
Monitoring Character Device Data Streams . . . . . 8-2
Device Monitor Support Limitations . . . . . 8-3
The OS/2 Monitor Mechanism . . . . . 8-5
The Character Device Monitor Process . . . . . 8-6
The Character Device Driver with Monitor Support . . . . . 8-7
The Relationship between a Character Device Driver and Its Monitors . . . . . 8-8
Registering and Terminating a Monitor . . . . . 8-8
Data Passing through a Monitor Chain . . . . . 8-9
Developing a Character Device Monitor . . . . . 8-11
OS/2 Monitor Function Calls . . . . . 8-11
DosMonOpen . . . . . 8-12
DosMonReg . . . . . 8-12
DosMonRead . . . . . 8-14
DosMonWrite . . . . . 8-16
DosMonClose . . . . . 8-17
Guidelines for a Character Device Monitor . . . . . 8-18
Monitor Buffers . . . . . 8-18
Monitor Data Records . . . . . 8-19
Positioning of Monitors in a Monitor Chain . . . . . 8-19
Monitor Thread Priorities . . . . . 8-20
Special Considerations for Character Device Monitors . . . . . 8-21
Performance . . . . . 8-21
Special Processing Requirements . . . . . 8-22
Monitor Termination . . . . . 8-22
Well-behaved Monitor Applications . . . . . 8-23
Monitor Problems and Solutions . . . . . 8-26
Type-ahead Characters . . . . . 8-26
Redirecting Data to Another Device . . . . . 8-29
Providing Monitor Support in a Character Device Driver . . . . . 8-31
The Monitor Dispatcher Device Helper . . . . . 8-31
MonitorCreate . . . . . 8-31
Register . . . . . 8-33
MonWrite . . . . . 8-33
MonFlush . . . . . 8-34
DeRegister . . . . . 8-35
Guidelines for a Character Device Driver . . . . . 8-36
Buffer Requirements . . . . . 8-36
Code Requirements . . . . . 8-36
Special Considerations for Character Device Drivers . . . . . 8-37
Device-specific Monitor Information . . . . . 8-38
Performance . . . . . 8-38
Device Driver Problems . . . . . 8-38
Sample Programs . . . . . 8-39
A character device driver (DEMODD.ASM) . . . . . 8-39
A character device monitor (MONITOR.ASM) . . . . . 8-39

**Chapter 9. I/O Privilege Level Code Segments (IOPL)** . . . . . 9-1

IOPL and the 80286 Protection Model . . . . . 9-1
IOPL versus Device Drivers . . . . . 9-3
System Services and IOPL Code Segments . . . . . 9-3
Guidelines for IOPL Code Segment Usage . . . . . 9-5
Building an IOPL Code Segment . . . . . 9-5
IOPL Sample Program (IOPL2.ASM) . . . . . 9-6

**Index** . . . . . X-1

# Chapter 1. Introduction

*Device drivers* send data to and receive data from physical devices; for example, from the keyboard or mouse and to the printer. Device drivers define the interface between the operating system or its applications and the physical device. They resolve device-independent I/O requests from the operating system and its applications with the device-dependent physical attributes of the device.

Request packets are the primary method of communication between the OS/2 Operating System™ and a device driver. OS/2™ receives I/O requests from applications and sends data in the form of request packets to the device driver. The device driver communicates with the device through interfaces, for example, the BIOS and ABIOS interface.

An application communicates with a device driver through one or more of the following interfaces:

| | |
|---|---|
| I/O subsystems - | Function calls that insulate an application from managing device I/O. |
| The file system - | Dynamic link function calls an application can use to redirect I/O. |
| The IOCTL interface - | A function call (DosDevIOCTL) that allows an application to send device-specific commands to a device driver. |
| IOPL code segments - | Privileged code segments from which an application can directly manipulate hardware interrupts and ports. |
| Character device monitors - | Dynamic link function calls that allow an application to view, insert or modify data passing through a device. |

An *I/O subsystem* is a set of dynamic link subroutines that service I/O requests for a single device resource. An application uses an I/O subsystem to communicate directly with the device driver, as well as with the device. An I/O subsystem uses the same methods that an application uses to communicate with a device driver and device.

I/O subsystems provided by OS/2 include the keyboard (KBD dynamic link function calls), mouse (MOU dynamic link function calls), and video (VIO dynamic link function calls) subsystems. The KBD, MOU and VIO subsystem function calls are described in the *OS/2 Version 1.2 Programming Guide* and the *OS/2 Version 1.2 Programming Reference*. The subsystem interfaces to the device drivers are described in this book in the sections describing the individual device drivers.

OS/2 and its device drivers manage the physical devices for applications running in OS/2 sessions. Only one application runs in an OS/2 session. OS/2 and its device drivers do not manage the physical devices for applications running in a Presentation Manager™ session. The Presentation Manager and Presentation Manager applications run in a single OS/2 session. The Presentation Manager defines its own sessions for each Presentation Manager application and manages the physical devices for those applications.

Because OS/2 and the Presentation Manager manage devices differently for each device, subsystem support varies in each environment. Device drivers support both environments. For example, a single keyboard and mouse device driver service requests from OS/2 as well as from the Presentation Manager. Keyboard, mouse, and video subsystem calls made by applications running in an OS/2 session are routed by OS/2 to the device drivers. Keyboard, mouse and video subsystem calls made by applications running in a Presentation Manager session are routed by the Presentation Manager to OS/2 and to the device drivers. This is illustrated by Figure 1-1 on page 1-2 and Figure 1-2 on page 1-3.

Additional subsystem support for printer, plotter, and video is provided for applications running under the Presentation Manager by *presentation drivers*. Presentation drivers are dynamic link routines that directly service I/O requests for devices. They operate with I/O privilege (IOPL).

---

™ Trademark of IBM Corporation

A separate video subsystem (the AVIO dynamic link function calls) is available for applications running under the Presentation Manager. The base video subsystem (the VIO dynamic link function calls) is not available to Presentation Manager applications.

Refer to the sections in this book describing the subsystem interfaces for each device for restrictions or limitations for applications running in the OS/2 or Presentation Manager environments.

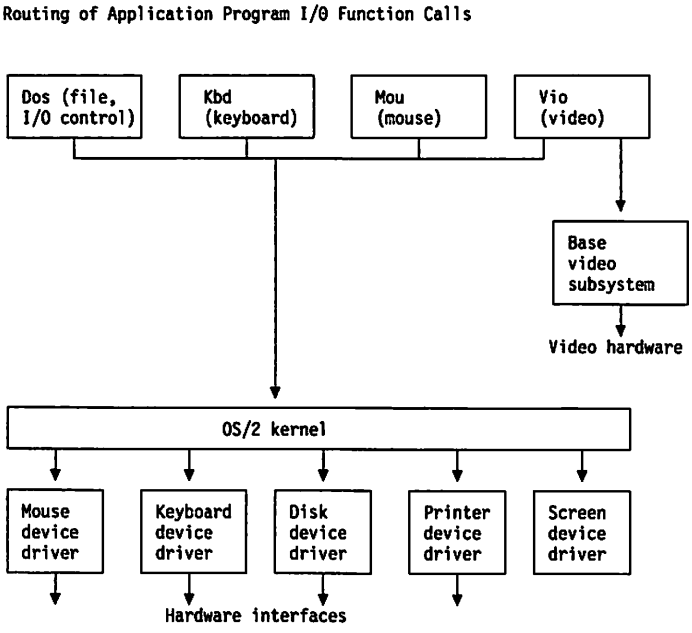



Figure 1-1. Applications running in an OS/2 session.

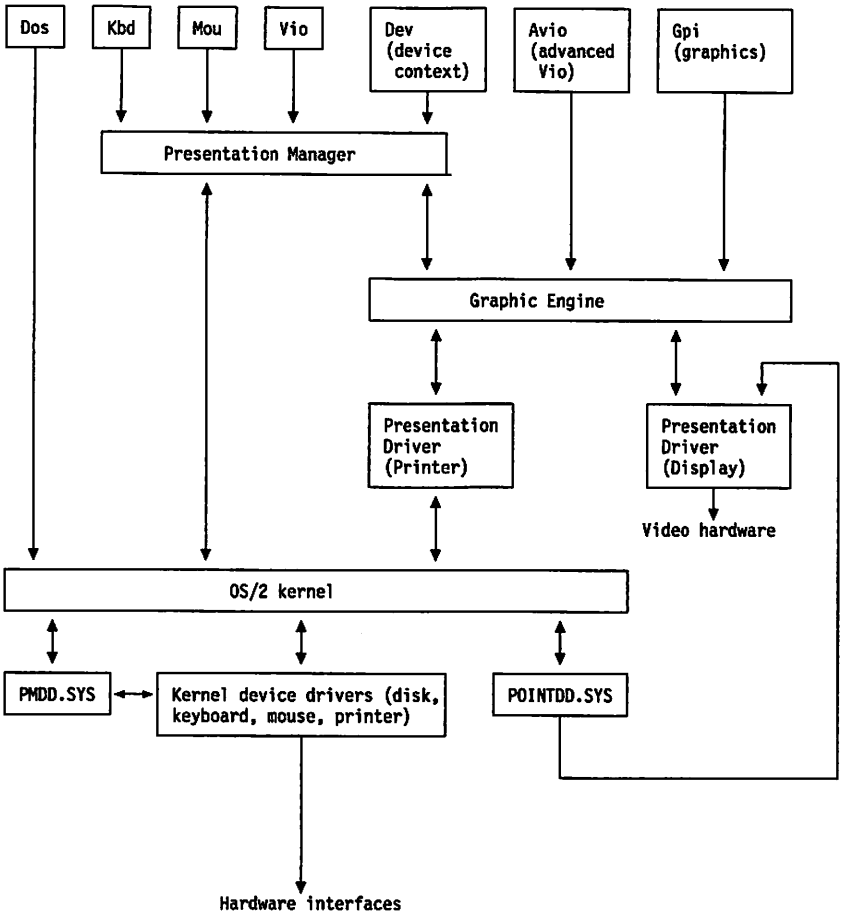



Figure 1-2. Applications running in a Presentation Manager Session

# OS/2 Device Drivers

The device driver interface to I/O devices is a low-level interface where data is passed to the device driver in a request packet. The device driver communicates with the device through, for example, the BIOS and ABIOS interfaces. Device drivers are loaded at privilege level 0, the most privileged level of the operating system.

Standard I/O devices are supported by base device drivers that are part of the OS/2 system. Additional, or replacement, drivers can be loaded to extend the control provided by the base drivers or to support non-standard I/O devices. Typical examples of loadable device drivers are ANSI.SYS and ANSI.DLL. These are loaded by DEVICE= statements in CONFIG.SYS and provide additional functions on the interfaces to the screen and keyboard.

Device drivers that are shipped with OS/2 include:

- Asynchronous Communication (RS-232C)
- Mouse Device Driver
- Pointer Draw Device Driver
- VDisk
- CLOCK$
- Console (Screen and Keyboard)
- Screen
- Keyboard
- EGA Register Interface

- Diskette
- Fixed Disk
- Printer.

OS/2 device drivers are bimodal. They service requests in the DOS™ mode and and the OS/2 mode. A detailed description of device support provided by each device driver is included in the first volume of this book. In addition, the first volume of this book provides a general description of OS/2 device drivers, including:

- Device driver structure and architecture
- Design considerations, including performance, nested interrupts, use of Advanced BIOS
- Interfaces to a device driver, including the request packet and generic IOCTL interfaces
- Subsystem interfaces to a device driver
- OS/2 services available for device drivers (that is, device helper services)
- Inter-device driver communication (device drivers calling other device drivers)
- Character device monitor support in character device drivers
- Use of IOPL code segments to directly manipulate the hardware.

A sample OS/2 device driver (DEMODD.ASM) is included on the sample diskette in the *OS/2 Version 1.2 Programmer's Toolkit.*

## 386 Programming Considerations

When executing on a system with a 386 processor, OS/2 now preserves the 386 extended registers across all context switches. These context switches include thread switch, process switch, and pre-emption due to a hardware interrupt. The registers preserved include:

- EAX
- EBX
- ECX
- ESI
- EDI
- EBP
- 32 Bit Flags Register.

These additional 16 bit 386 segment registers are also preserved:

- FS
- GS.

This does not imply that segments of greater than 64K are supported. It does imply that the high order word of the following registers will be destroyed on a context switch:

- ESP
- EIP

# OS/2 Presentation Drivers

The Presentation Manager I/O interface for output devices is a high-level interface. This interface is similar to the API call interface, which uses the program stack to communicate with, or pass parameters to, the presentation drivers. These drivers are special purpose I/O routines, operating with I/O privilege at privilege level 2. Their main function is to process function calls made by the Presentation Manager on behalf of Presentation Manager applications. Printer and plotter presentation

™ Trademark of IBM Corporation

drivers communicate with OS/2 device drivers through file system emulation calls. Screen presentation drivers interface directly to the hardware.

Presentation drivers are dynamic link library modules that are supplied as files and identified by the extension .DRV. When the Presentation Manager is initialized, the screen presentation driver is loaded and enabled automatically. Other presentation drivers, for example, the printer and plotter presentation drivers, are loaded and enabled when an application calls the DevOpenDC function to open the device.

Presentation drivers service requests from applications running in Presentation Manager sessions in the OS/2 mode only. The second volume of this book provides a general description of OS/2 Presentation Drivers, including:

- The presentation driver interface, including a detailed description of the calling conventions for this interface
- System services available to a presentation driver.

Output data and requests for information are passed to the presentation driver as function calls to the presentation driver interface. The definition of the call interface in this book is given in terms of the codes and data passed to the presentation driver interface through the program stack.

Header and include files are shipped with OS/2 to provide extensive support for building presentation drivers written in either the C or assembler languages. These files define names for the functions and values and the data structures used by the calls.

# Chapter 2. Device Driver Overview

Device drivers written for DOS are synchronous and often non-interrupt driven. Because DOS is a single-task operating system, this presents no problem. A program cannot proceed until the I/O has been completed, so it is acceptable for the device driver to hold the CPU until the I/O is complete.

OS/2, however, is a multitasking operating system. It must be able to assign the CPU to tasks that are not waiting for I/O. Therefore, device drivers written for OS/2 must surrender the CPU while they are waiting for I/O completion. Consequently, OS/2 device drivers must be interrupt-driven.

In addition, OS/2 provides a DOS mode to support DOS applications, so OS/2 device drivers must support device I/O requests in the DOS mode. This means that OS/2 device drivers must handle I/O requests issued by the applications running in the DOS mode and may need to interlock DOS mode ROM BIOS device I/O with OS/2 mode device I/O. Performance considerations also require that device drivers are able to handle hardware interrupts without the overhead of a mode changing method. OS/2 device drivers must therefore be bimodal, that is, execute in both the OS/2 mode and the DOS mode.

**Note:** There is no facility to define a protect-mode-only device driver.

The basic characteristics of the OS/2 device driver are:

- Support of a multitasking environment
- Execution in both the DOS mode and the OS/2 mode.

## Types of Device Drivers

There are two types of device drivers:

- Character device drivers
- Block device drivers.

Character device drivers manage I/O on character-oriented devices. These devices perform I/O on a character-by-character basis. A character device has a name like SCREEN$, KBD$, LPT1, or COM1. A character device driver can support more than one device by having multiple linked device headers where each header indicates a different name.

Block device drivers support block-oriented devices. These devices perform I/O on a block of data, typically through Direct Memory Access (DMA). Applications request I/O on block-oriented devices through a file system. Consequently, block device drivers do not have names. Instead, a block device driver is assigned one or more drive letters. A block device driver can support multiple devices (or units), so a drive letter is assigned for each unit (or device). A block device driver specifies the number of devices that it supports when it is called to initialize. Refer to the INIT device command, "0H / INIT Initialize Device Driver" on page 4-5.

The order in which the block device drivers appear in the CONFIG.SYS configuration file (through the DEVICE= command) determines the order in which they receive drive letters.

## Installation of External Loadable Device Drivers

The user may install external loadable device drivers through the system utility program DDINSTAL.EXE. (See *OS/2 Version 1.2 Using the System* for a description of this utility.) A Device Driver Profile (a file with a .DDP file name extension) must be provided by the device driver author to control the installation of the device driver using DDINSTAL.EXE. A diskette containing device drivers with their corresponding Device Driver Profiles is called a Device Support Diskette. DDINSTAL.EXE will install all Device Driver Profiles on a Device Support Diskette. When DDINSTAL.EXE has finished installing the Device Support Diskette, it will prompt the user to restart the system, causing the updated CONFIG.SYS file to take effect. DDINSTAL.EXE is not intended to install Presentation drivers, which are installable from the Control Panel.

## Device Driver Profile

The device driver profile is comprised of two sections: the :CONFIG section and the :FILES section. Each section is optional and may appear any number of times, but the profile must contain at least one section. Blank lines are ignored and * begins a comment which is terminated by the end of the line.

### :CONFIG

The :CONFIG section describes the CONFIG.SYS statements required for loading the device driver when the system is initialized. Each line in this section is added to the end of C:\CONFIG.SYS. These lines should be valid CONFIG.SYS statements; no checks are made for conflicting or duplicate entries. Because the complete line is appended to the CONFIG.SYS file, lines in this section should not be commented.

### :FILES

The :FILES section describes the files to be copied to the fixed disk by DDINSTAL.EXE. Each line in this section contains the source name and destination name of a file to be copied, separated by spaces. The default drive and path for the source name is A:\ and the default drive and path for the destination name is C:\. The date, time and attributes of the file are preserved during the copy. If the destination path does not exist, it is created before the file is copied.

If the destination name is an existing file in use, such as an open dynamic link library or a program that is running, DDINSTAL.EXE will create a temporary directory, C:\OS2\INSTALL\DDITEMP. All files for which this occurs will be copied to the temporary directory and a temporary profile, DDITEMP.DDP, will be created, containing the names of the files and all the lines from the :CONFIG sections. After DDINSTAL.EXE has processed the remainder of the Device Support diskette, it will prompt the user to restart the system from the Installation diskette. The Installation diskette will automatically execute DDINSTAL.EXE to continue the installation from the temporary directory. When the installation is complete, DDINSTAL.EXE removes the temporary directory and prompts the user to remove the Installation diskette and restart the system, causing the updated CONFIG.SYS file to take effect.

## Sample Device Driver Profile

The format of a Device Driver Profile is illustrated in Figure 2-1 on page 2-3.

```
* This is a Device Driver Profile for a fictitious piece
* of hardware. The software consists of a device driver,
* an I/O subsystem and test programs.

*     The Device Driver

:CONFIG       * This section contains the CONFIG.SYS
          * statements for the device driver
DEVICE=c:\devdrvr\devdrvr.sys

:FILES        * This section contains the files for
          * the device driver
devdrvr.sys c:\devdrvr\devdrvr.sys     * if the directory
                              * doesn't exist,
                              * then create it

*     The I/O Subsystem

:CONFIG       * This section contains the CONFIG.SYS
          * statements for the I/O subsystem
* The IOPL statement specifies the module name in the
*  DLL and allows it to run at ring 2.
IOPL=devdrvr

:FILES        * This section contains the files for
          * the I/O subsystem
devdrvr.dll c:\os2\dll\devdrvr.dll     * copy to the DLL
                              * directory

*     The Test Programs

:FILES        * This section contains the files for
          * the test programs
devdrvr1.exe c:\devdrvr\devdrvr1.exe   * test program 1
devdrvr2.exe c:\devdrvr\devdrvr2.exe   * test program 2
```

Figure 2-1. A Sample Device Driver Profile

# Application I/O to Devices

The primary interface to a device driver is the OS/2 request packet interface. File I/O and IOCtl requests from OS/2 mode applications and INT 21H requests from DOS mode applications are translated into request packets that are sent to the device driver for processing. DOS mode software interrupts are directly processed by the bimodal OS/2 device driver.

An OS/2 device driver is the standard interface to a device. OS/2 device drivers manipulate the devices on behalf of all applications and subsystems. However, when the device driver interface is insufficient, both OS/2 mode and DOS mode applications may:

- Access a device directly by sending data to or receiving data from a specific port address through IN/OUT instructions.
- Manipulate the state of the hardware interrupts (disable/enable interrupts) through CLI/STI instructions.

Any DOS mode application may directly access a device from its code segment. OS/2 mode applications and subsystems may directly access a device only through IOPL code segments.

Figure 2-2 on page 2-4 illustrates how DOS mode and OS/2 mode applications access a device driver.

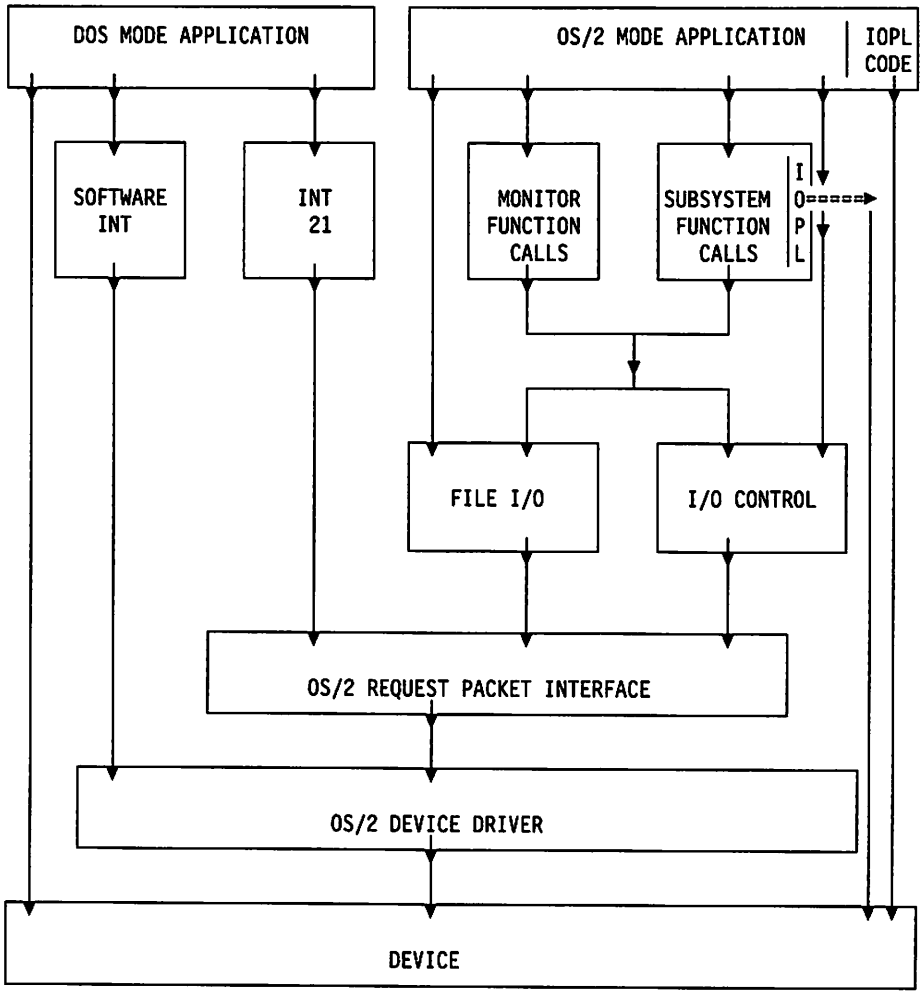



Figure 2-2. Application I/O to a Device Driver

## I/O Support for the OS/2 Mode

An OS/2 application can perform I/O requests using the interfaces described in Table 2-1.

Table 2-1. Accessing Device Drivers

| Interface | Example | Advantage |
|---|---|---|
| File system | DOS dynamic link function calls | An application can redirect the I/O. |
| Subsystem | VIO, KBD, MOU function calls | Insulates an application from managing device I/O. |
| IOCtl | DosDevIOCtl commands | The subsystem or application uses IOCtl commands to send device-specific control commands to a device driver. |
| Device Monitors | OS/2 dynamic link monitor function calls | The application can directly insert, view or modify data passing through a device. |
| IOPL code segments | OS/2 function calls to request permission to use I/O ports and manipulate the state of the hardware interrupts | The application can directly access a device's I/O ports. |

The File system interface consists of the file I/O function calls. Both OS/2 and DOS applications use the DOS dynamic link calls for file I/O. DOS applications running in the DOS mode issue INT 21H File I/O function calls. The file I/O function calls are used primarily to perform I/O on block devices like fixed disks.

Some file I/O function calls can be used to perform I/O on character devices. These calls include:

- DosOpen
- DosClose
- DosRead
- DosReadAsync
- DosWrite
- DosWriteAsync.

The advantage of using the file I/O function calls to perform I/O on a character device is that the application can redirect the I/O. The application performs I/O to a handle (file or device) that it obtained from opening the named resource passed to it.

A subsystem interface shields an application from having to manage device I/O. The file system is an example of a subsystem. The file system allows the application to view a file as a logical sequence of sectors, thus shielding the application from having to manage the physical locations of the data on the disk media. Other subsystem examples include the VIO, KBD, and MOU subsystems that provide I/O services for video, keyboard, and mouse devices respectively.

The IOCtl interface is the mechanism that an application or subsystem uses to send device-specific control commands to the device driver. The IOCtl mechanism is DosDevIOCtl for new applications and the INT 21H IOCtl request for DOS applications. I/O commands can be sent to both block and character device drivers. The application or subsystem must first obtain the device handle by doing an open on the device name.

Character Device Monitors provide a mechanism for applications or subsystems to monitor all characters passing through a character device driver. This mechanism allows one or more registered processes to remove, insert, or modify data passing through a character device. This mechanism is supported by five dynamic link function calls:

- DosMonOpen
- DosMonReg
- DosMonClose
- DosMonRead
- DosMonWrite.

See Chapter 8, "Character Device Monitors" in this book for more information. An OS/2 application may directly access a device through service routines defined in its own IOPL code segments. Permission to access a device is requested by calling:

- DosPortAccess
- DosCLIAccess

**Note:** A device driver is the standard way to provide service to and from OS/2 applications. IOPL code segments should be used only when the standard device driver interface is not sufficient.

See Chapter 9, "I/O Privilege Level Code Segments (IOPL)" in this book for more information.

## I/O Support For The DOS Mode

A device driver is responsible for managing I/O for its device. Because many devices must be accessible from both the DOS mode and the OS/2 mode, an OS/2 device driver must manage its device across both modes of operation. Examples of such cross-mode devices are disk, keyboard, screen, mouse and printer.

Device I/O in the DOS mode is performed in one of three ways:

- DOS INT 21H interface
- ROM BIOS interface
- Direct access to the device.

I/O using the DOS INT 21H interface is transformed into a request packet, which the device driver receives. (Similarly, I/O using the OS/2 dynamic link function calls is transformed into a request packet.) Because the request packet interface for new device drivers is standard across modes, it poses no problem for the device driver in managing its device.

Because they are bimodal, OS/2 device drivers will service INT 21H functions that create standard I/O request packets. This level of support exists in all OS/2 device drivers.

**Note:** Installable OS/2 device drivers are not required to support ROM BIOS software interrupt handling. This level of support is optional. Lack of support for ROM BIOS will restrict the number and type of DOS applications able to access that device.

**Note:** OS/2 device drivers are not required to share control of the device with DOS applications that directly access the device hardware. Such DOS applications may not be fully supported in the OS/2 DOS mode.

I/O using the ROM BIOS poses some problems for an OS/2 device driver. The OS/2 device driver must intercept the ROM BIOS software interrupt (by setting the vector with the DevHlp SetROMVector — see "SetROMVector Set DOS Mode Software Interrupt Vector" on page 5-59) and interlock ROM BIOS operations on its device in two ways:

- Serialize access to the device.

  Serialize by using semaphores to indicate when the device is busy with a request (and consequently cannot accept/tolerate a request from ROM BIOS).

- Protect critical sections of ROM BIOS execution from suspension.

  Suspension occurs when a user switches away from the application in the DOS mode to an application in the OS/2 mode. This causes the DOS mode to be suspended in the background. However, some I/O processing cannot tolerate being suspended. Specific examples are the printer (BIOS INT 17H), disk (BIOS INT 13H), and screen (BIOS INT 10H). It is the responsibility of the OS/2 device driver to intercept the appropriate ROM BIOS interrupt and issue the DevHlp function call, ROMCritSection, to protect the ROM BIOS critical section of execution.

  **Note:** When the OS/2 device driver issues ROMCritSection to enter a ROM BIOS critical section, the user is not able to switch away from the application in the DOS mode to an application in the OS/2 mode. This poses potential problems for the user. For example, if a DOS mode terminate-and-stay-resident program takes control while the CPU is executing the ROM BIOS, the time spent in the ROM BIOS critical section will be longer. The worst case is that the terminate-and-stay-resident application is interactive, never allowing the OS/2 device driver to issue the exit from critical section and never allowing the user to

switch away from the application in the DOS mode until the user terminates the application.

Application I/O using direct access to a device driver's device poses the same problem for the device driver under OS/2 as it does for a device driver operating under DOS. If device state information is critical or if device I/O must be serialized, then a device driver can choose not to access the full function of the device. Control of the device, in this case, would be shared between the application and the device driver.

# Device Driver Initialization

Device driver initialization occurs during system initialization. During system initialization, base device drivers are pre-loaded with the operating system. Installable device drivers are loaded during CONFIG.SYS processing, by way of the DEVICE= configuration command. After a device driver has been loaded, it will be called at its strategy routine entry point with the request packet for the INIT command.

OS/2 device drivers have the following characteristics at INIT-time:

- Occupy memory below 640 KB
- Initialize in the OS/2 mode
- Can use Advanced BIOS services
- Have IOPL at all times
- Can service hardware interrupts
- Can use timer services
- Can use some OS/2 dynamic link function calls.

During CONFIG.SYS processing, each DEVICE= command is processed on a first-come-first-served basis:

- The DEVICE= device driver file image is loaded into low memory.
- The check for DEINSTALL of a previously initialized device driver is performed.
- If DEINSTALL is successful, the newly loaded device driver is initialized.

Installable OS/2 device drivers initialize in the OS/2 mode. The device driver strategy routine will run under the thread of the System Initialization Process, at privilege level 3, with I/O privilege. Because of the special system process, the installable device driver is allowed to make dynamic link function calls only at INIT-time.

## Device Driver INIT-Time Function Call Summary

Those dynamic link function calls available to a device driver at INIT-time include:

| | |
|---|---|
| **DosBeep** | Generate Sound From Speaker |
| **DosCaseMap** | Perform Case Mapping |
| **DosChgFilePtr** | Change (Move) File Read/Write Pointer |
| **DosClose** | Close File Handle |
| **DosDelete** | Delete File |
| **DosDevConfig** | Get Device Configuration |
| **DosDevIOCtl** | I/O Control for Devices |
| **DosFindClose** | Close Find Handle |
| **DosFindFirst** | Find First Matching File |
| **DosFindNext** | Find Next Matching File |
| **DosGetEnv** | Get Address of Process Environment String |
| **DosGetMessage** | System Message with Variable Text |
| **DosOpen** | Open File |
| **DosPutMessage** | Output Message Text to Indicated Handle |
| **DosQCurDir** | Query Current Directory |
| **DosQCurDisk** | Query Current Disk |
| **DosQFileInfo** | Query File Information |
| **DosQFileMode** | Query File Mode |
| **DosRead** | Read from File |
| **DosSMRegsIterDD** | Register session switch notification |
| **DosWrite** | Synchronous Write to File. |

For a detailed description of each function call, see the *OS/2 Version 1.2 Programming Reference.*

**Note:** Because device driver initialization is invoked from the strategy routine, a device driver must not issue a DosExit function call. Instead, the device driver should return the INIT request packet by setting the packet's return status and performing a FAR RET to the kernel.

In addition to the set of dynamic link function calls, certain DevHlp services are available to device drivers at INIT-time. These functions are listed in the chart of DevHlp services. See Table 5-1 on page 5-2.

For more information on device driver initialization, see the description of the device driver strategy command "0H / INIT Initialize Device Driver" on page 4-5.

## Replacing Character Device Drivers

OS/2 character device drivers can be replaced. For system character device drivers, the appropriate bits in the attribute field of the device driver header and the name of the character device driver must match.

When the new device driver is loaded, the attribute field and name are used to determine if the new device driver is attempting to replace a driver already installed. If so, the previously installed device driver is requested to DEINSTALL the indicated device. If the already installed device driver refuses the DEINSTALL command, the new device driver is not allowed to initialize. If the already installed device driver performs the DEINSTALL, the new device driver is initialized.

**Note:** The DEINSTALL command is needed to allow a device driver to relinquish its interrupt vectors and its allocated physical memory.

## Compatibility with Previous-Level Device Drivers

The term "previous-level" is used here to indicate DOS levels below DOS 3.3.

Not all previous-level DOS device drivers can be allowed to run in the DOS mode. The supported set of previous-level device drivers has the following restrictions:

- Previous-level block device drivers are not permitted in the DOS mode. Block device drivers must be written to the OS/2 interfaces.
- Only a limited set of previous-level character device drivers can be supported by OS/2. In order to run in the DOS mode, a previous-level character device driver must conform to the following rules:
  - The character device driver cannot have a hardware interrupt handler: its device must be a polled device rather than an interrupt-driven one.
  - The character device driver can be called by OS/2 in the DOS mode with all of the packets supported by character devices:

    0 — INIT
    3 — IOCtl input
    4 — INPUT (read)
    5 — NON-DESTRUCTIVE INPUT NO WAIT
    6 — INPUT STATUS
    7 — INPUT FLUSH
    8 — OUTPUT
    9 — OUTPUT with verify
    10 — OUTPUT STATUS
    11 — OUTPUT FLUSH
    12 — IOCtl output
    13 — DEVICE OPEN
    14 — DEVICE CLOSE
    16 — GENERIC IOCtl

    Receipt of these packets is limited by the same requirements on the attribute field of the device header as in DOS 3.3. For example: IOCtl bit in the device header must be 1 to receive IOCtl requests.

The side effect of running a previous-level character device driver is that its device may be used only in the DOS mode. Only applications running in the DOS mode can perform I/O to this device. Applications in the OS/2 mode are not allowed to access the device.

Certain devices cannot be exclusive to the DOS mode. Character devices in this category include:

- Mouse
- Clock.

Consequently, previous-level clock and mouse device drivers cannot be supported.

## Initialization of Previous-Level Device Drivers

Previous-level character device drivers are installed in the same manner as they were under DOS. The device driver program file is specified in the configuration command, DEVICE=.

Previous-level device drivers are loaded and initialized in DOS mode. The rules for replacing previous-level character device drivers are the same. The replacement is guided by the name and attributes of the device driver. The functions that can be performed at initialization are more restrictive than for DOS 3.3. No INT 21H functions can be performed from the device driver initialization code.

# Device Driver Architecture

The file image of a device driver is shown in Figure 2-3.

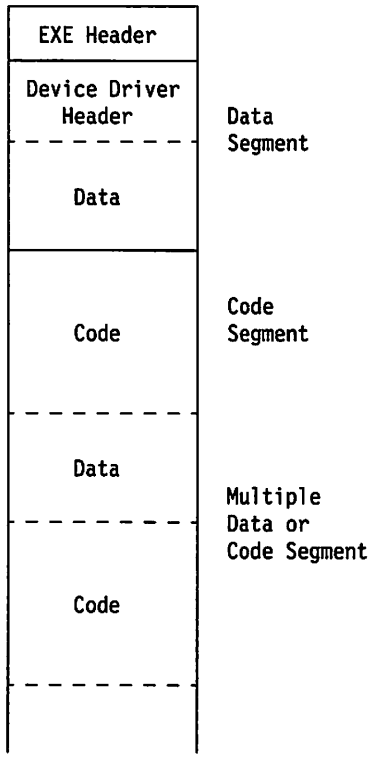



Figure 2-3. File Image of a Device Driver

The data segment must be the first segment after the .EXE file header. This allows the device header to be located immediately after the .EXE file header because the device header must be located at the beginning of the file.

**Note:** Multi-segmented device drivers that support monitors must be organized so that the monitor notification routine resides in the first code segment or the fixed low memory segment.

## Device Driver Program Model

An OS/2 device driver may have multiple code and data segments. The first segment in the device driver must be a data segment, and the second must be a code segment. These two segments will be fixed in low memory. All other segments in the driver reside in high memory. By default, only the first two segments are kept in the system after device driver initialization. OS/2 assumes that every other segment is needed for initialization only, unless it is marked as permanent in the header of the .EXE file.

A permanent segment remains after initialization and is assumed to be movable and swappable, but not discardable. A driver may lock a permanent segment using the appropriate Device Helper routine if it is necessary to prevent it from being swapped out or moved. IOPL is available at privilege level 3 during device driver installation. Device drivers, therefore, have no need to create privilege level 2 segments. The same bits in the .EXE file that specify that a .DLL segment is privilege level 2 specify that a device driver segment is permanent. The device driver developer uses the SEGMENTS directive and its IOPL option in the linker .DEF file to mark those segments that are to be kept after initialization. Since the first two segments are always kept, they do not have to be marked in this fashion.

The first code segment must contain code that can be executed in either real or protected mode. All other code segments are kept in high memory, and, therefore, can be executed only while the processor is in protect mode. Similarly, when the processor is in real mode, only data in the first

data segment may be accessed directly. (There is, however, a way of accessing data in high memory while in real mode - see the PhysToVirt device helper routine.) When passing control to a device driver, OS/2 will set CS and DS to reference the first code and data segments, respectively, using values that are appropriate to the processor's current mode of execution.

**Note:** Device drivers that must run under OS/2 Version 1.0 are limited to a single code and a single data segment in low memory after initialization. There is no support in Version 1.0 for permanent segments in high memory.

The device driver executable image does not contain a stack segment. A stack is provided by the system. However, at initialization, a device driver may indicate to the system its stack usage by calling the RegisterStackUsage device helper routine for each IRQ it services.

## Device Driver Header

The device driver's first data segment must contain a *device header* as the first item. The structure of a device header is shown in Table 2-2.

Table 2-2. Device Header structure

| Field | Length |
|---|---|
| Pointer To Next Header | DWORD |
| Device Attribute | WORD |
| Offset To Strategy Routine | WORD |
| Offset to IDC Entry Point | WORD |
| Name or Units | 8 BYTES |
| Reserved | 8 BYTES |

Where:

**Pointer to Next Device Header Field**

is the pointer to the next device header. This is set by OS/2 at the time the device driver is loaded. This field should be set to -1.

**Note:** For a character device driver that supports multiple devices, the data segment contains a device driver header for each device. These headers must be linked together and the last header must be set to -1. The first word is an offset and the second word is the segment.

**Device Attribute Field**

describes the characteristics of the device driver to the system.

The format of the OS/2 device attribute field is:




Figure 2-4. Device Attribute field

The attributes are:

| *Bit#* | *Meaning* |
|---|---|
| 15 | Set if the device driver operates in character mode. |
| 14 | Set if the device driver participates in inter-device driver communication (IDC). Bit 14 indicates that the Offset to the IDC Entry Point in the device driver header is set. |
| 13 | Set if non-IBM block format, (block device drivers only). Set for output-until-busy support, (character device drivers only). |

12 Set if support shared device access checking (character devices).
11 Set if support removable media (block devices) or device open/close (character devices).
10 Reserved = 0
9-7 Function level where 001 = OS/2 device driver.
010 = Supports DosDevIOCtl2 request packets.
6 Reserved = 0
5 Reserved = 0
4 Reserved = 0
3 Set if CLOCK device.
2 Set if NULL device.
1 Set if standard output device (STDOUT).
0 Set if standard input device (STDIN).

**Bit 15** Bit 15 is the device type bit. Use bit 15 to tell the system if the device driver is a block or character device. For block device drivers, bit 15 is 0. For character device drivers, bit 15 is 1.

**Bit 14** Bit 14 is the IDC bit, indicating that other device drivers may communicate with it. The Offset to the IDC Entry Point is included in the device driver header.

**Bit 13** For block device drivers, bit 13 indicates the method the driver uses to determine the media type.

If a block device driver uses information in the BIOS Parameter Block (BPB) to determine the media type, bit 13 should be set to 1. If the device driver uses the media descriptor byte to determine the media type, bit 13 should be 0.

**Bit 12** Bit 12 is the shared bit. It is set if the device name is NOT to be protected by sharer. (Bit 12 has no meaning for block device drivers, must be 0.)

If clear (default), file system sharing rules DO NOT apply to the device, and it is the responsibility of the device driver to provide contention control.

If set, file system sharing rules DO apply to the device, just like they apply to any other file system name. In addition, any given physical device may have only one logical name. (Devices cannot have aliases.)

**Bit 11** For block device drivers, bit 11 is the removable media bit. If set, this bit indicates that the device driver handles removable media.

For character device drivers, bit 11 is the open/close bit. If set, this bit indicates that the device driver must receive OPEN AND CLOSE request packets.

**Bit 9-7** Bits 9-7 indicate the function level where 001 = OS/2 device driver.

**Bit 3** Bit 3 is the clock device bit. It is used by a character device driver to indicate the system clock device.

**Bit 2** Bit 2 is the NULL attribute bit. It is used by character devices only. Use bit 2 to tell OS/2 if your character device driver is a NULL device. Although there is a NULL device attribute bit, you cannot reassign the NULL device. This attribute exists so OS/2 can tell if the NULL device is being used.

**Bits 1 and 0** For character devices, bits 1 and 0 are the standard input/standard output bits. Use these bits to tell OS/2 if your character device driver is the new standard input or standard output device.

**Offset to Strategy Routine Field**

contains the offset from the start of the code segment to the strategy entry point. OS/2 uses this offset to call the strategy routine. The pointer is a word value contained in the device header.

**Offset to the IDC Entry Point**

is the offset from the start of the device driver's code segment to the entry point which is callable by other device drivers.

**Name/Units Field**

contains the name of a character device supported by the character device driver or the number of units supported by the block device driver.

For a character device driver, the name of the device must be uppercase ASCII characters and must be left-justified with the remaining space set to blanks. The device name is used by applications to identify the device for I/O. A character device driver should consider the following rule when selecting a device driver name:

*A device name takes precedence over a filename in a DosOpen function call. This means that files cannot have the same name as a character device. The DosOpen function call will always open the device rather than the file.*

**Note:** To avoid such conflicts with filenames, a character device driver should choose a character string with some unusual character such as a $ sign.

For a block device driver, the number of units can be placed in the first byte. This is optional because OS/2 will fill this field during device driver initialization.

For character device drivers using ABIOS, the device name represents a single device identified by the Logical ID (LID). For block device drivers using ABIOS, the number of units is equivalent to the number of devices (or units) under the LID.

## Device Driver Components

Figure 2-5 illustrates the components of a device driver.

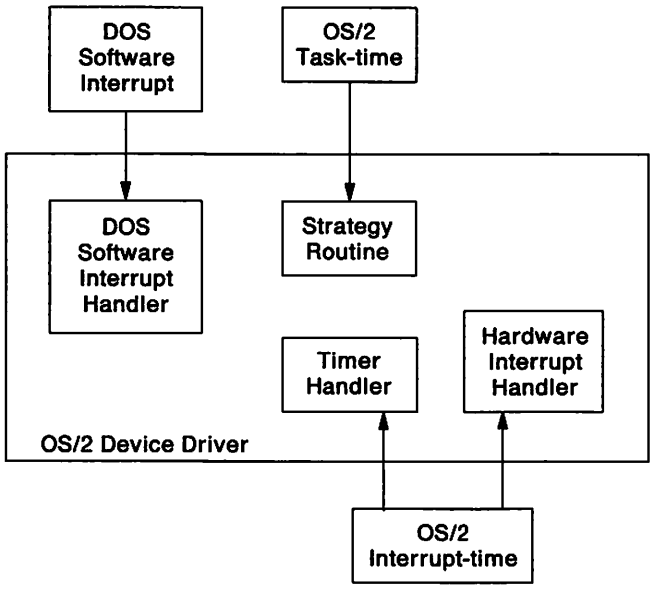



Figure 2-5. Components of a Device Driver

Deciding what components to include in your device driver depends on two things:

- The complexity of your task
- How the application will perform I/O to its device.

**Note:** The strategy routine is required. The hardware interrupt handler is recommended. The timer handler and software interrupt handler are optional.

A device driver is comprised of one or more of the following:

1. Strategy Routine (Required)

   The strategy routine is called to handle I/O requests through a request packet interface with the OS/2 kernel. The pointer to the request packet (ES:BX) is bimodal.

   The strategy routine executes at task-time as a result of an application I/O request. Because application I/O requests can come from new OS/2 mode applications active in the protect mode

of the processor and from DOS mode applications active in the real mode of the processor, the strategy routine must not be dependent on the mode in which it is called.

The strategy routine follows the far call/return model. When it completes processing, it issues a far return to the kernel. By convention, OS/2 preserves any registers used by the strategy routine.

The request packet interface to device drivers is described in Chapter 4, "Device Driver Strategy Commands."

2. Hardware Interrupt Handler (Recommended)

   The hardware interrupt handler is called as a result of a hardware interrupt and executes at interrupt time. The hardware interrupt handler must not be dependent on the processor mode in which it is called.

   The hardware interrupt handler follows the far call/return model. When it completes processing, it issues a far return to OS/2. It must clear or set the CF (Carry Flag) to indicate whether it owns the hardware interrupt. By convention, OS/2 preserves any registers used by the hardware interrupt handler.

3. Timer Handler (Optional)

   The timer handler is called as a result of a periodic clock tick and executes at interrupt time. The timer handler manages timeouts and is similar to the INT 1CH user time feature of ROM BIOS. The timer handler must not be dependent on the mode in which it is called.

   The timer handler follows the far call/return model. When it has completed processing, it issues a far return to OS/2. The timer handler must save and restore any registers it uses.

4. Software Interrupt Handler (Optional)

   The software interrupt handler is called directly by a software interrupt. Software interrupts can be issued only in the real mode. The software interrupt handler, therefore, executes only in real mode and follows the interrupt/return model.

   Typically the software interrupt handler is used to intercept ROM BIOS software interrupts to serialize device access between the protect mode and the real mode. The handler also prevents the BIOS service from being suspended when the user attempts to switch away from a real mode application to a protect mode application.

## OS/2 Device Driver Contexts

There are four contexts (modes) in which OS/2 device drivers operate. They are:

- Kernel Mode

  The OS/2 kernel calls the device driver strategy routine for task-time operations. (Task time is a generic term that refers to executing code as a thread within a process.) The strategy routine will execute as a thread within a process. The strategy routine will not be preempted by a task switch but may be interrupted by incoming hardware interrupts. Kernel mode applies to both the DOS mode and the OS/2 mode.

- Interrupt Mode

  The OS/2 kernel calls the device driver interrupt-time components, the hardware interrupt handler, and the timer handler. (Interrupt time is a generic term that refers to executing code as a result of an interrupt; the thread of execution does not belong to a process.) The hardware interrupt handler and the timer handler will not execute code as a thread belonging to a thread specific process; the thread of execution results from a hardware interrupt. Interrupt mode applies to both the DOS mode and the OS/2 mode.

- User Mode

  The device driver software interrupt handler is called, and applies only to the DOS mode. The software interrupt handler is invoked by a software interrupt. In this mode, the device driver software interrupt handler may be preempted by a task switch.

- INIT Mode

The device driver strategy routine is called with a request packet containing the INIT command. The initialization code runs in the OS/2 mode at the application privilege level (privilege level 3) with I/O privilege. A limited set of dynamic link function calls are available for use, as well as a portion of the device helper (DevHlp) function calls. This is discussed in the section "Device Driver Initialization" on page 2-7.

## OS/2 Device Driver Operations

The following example shows the interaction between the strategy routine and the hardware interrupt handler in the processing of an I/O request.

The handling of an I/O request begins with OS/2 calling the strategy routine entry point with a request packet. The strategy routine checks the validity of the I/O request. If valid, the strategy routine may place the request on a work queue for the device, using the DevHlp functions for request queue management.

If the device is currently idle, the strategy routine starts the request at the device. The strategy routine may then wait for the device driver interrupt by suspending its thread of execution with the DevHlp function BLOCK.

When the device interrupt occurs, the hardware interrupt handler checks the request to see if it has been completed. If the request has not been completed, the hardware interrupt handler continues the request at the device. If the request has been completed, the hardware interrupt handler sets the return status in the request packet. The hardware interrupt handler may remove the completed request packet from the work queue and start the next request at the device. If the strategy routine is waiting for the device interrupt (which is blocked), the hardware interrupt handler can wake up the strategy routine with the DevHlp RUN.

In this example, the strategy routine queues the requests and only initiates the I/O if the device has been inactive. The hardware interrupt handler starts requests as they reach the head of the work queue. The thread context, in which the device driver determines that a particular request is complete, is not necessarily the same thread context in which the device driver received the request. This is particularly true for the interrupt-time components of the device driver. For example, the address of a user buffer passed to the device driver when the request was issued belongs to a specific local descriptor table (LDT), which may not be the current LDT when the request ends. The device driver can accommodate this by storing the buffer address as a 32-bit physical address.

The device driver strategy routine is called by OS/2 with a pointer to the request packet. The pointer to the request packet is bimodal; that is, the pointer is valid in both the DOS mode and the OS/2 mode. Any addresses passed in the request packets for read/write requests are passed as 32-bit physical addresses (normalized). Therefore, the device driver does not need to lock or convert the addresses into physical addresses. The device driver only needs to lock addresses that it receives from a source other than OS/2, as in the case of a process passing an address by way of a generic IOCtl.

The multitasking environment dictates that the components of the device driver must be capable of handling requests simultaneously. This means that the components of the device must relinquish execution whenever possible. The device driver relinquishes control at task-time by BLOCKING, YIELDING, or referencing a segment which had been swapped out; OS/2 will not preempt a thread in the device driver. However, once the device driver releases its execution, OS/2 can call the device driver with a new request. Once the strategy routine BLOCKS, YIELDS, or references a swapped-out segment, its thread of execution can be called with a new request under the context of a different thread.

While the strategy routine can assume that it will not be preempted by other task-time instances, it must protect itself against its own interrupt-time components. It should disable interrupts when checking if the device is active and when examining the device queue. The interrupt-time components will only be preempted by other higher priority interrupts.

One component of the device driver may be preempted, the software interrupt handler. The software interrupt handler is invoked by a software interrupt in the DOS mode. It can be preempted by background OS/2 mode threads which are scheduled to execute and which may issue I/O requests causing other components of the device driver to be invoked.

## Request Packet Queue Management

The strategy routine can either queue a request packet or process it immediately. Typically, only read requests and write requests need to be queued because the device is busy. Other types of requests can usually be handled immediately by the strategy routine.

A block device driver, such as the disk device driver, can process queued requests in any order. For instance, the block device driver can choose to sort the requests to optimize device access time. A character device driver must always handle queued requests in the order it received them; otherwise, mixed output could result.

The request packet queue is really a linked list. The request packet contains a linkage field which allows the packets to be chained together.

The device driver can manage its work queue of request packets with the DevHlp functions for Request Queue Management.

To use the Request Queue Management DevHlp services, the device driver must allocate a DWORD variable as a queue header, with one header for each queue. The DWORD variable must be initialized to zero to indicate an empty linked list or the end of the linked list.

The DevHlp services use the queue header to identify a specific linked list of request packets and will set the header to the first request packet in the list. The linkage field in the request packet then chains the request packet to another request packet.

Because the pointer to the request packet is bimodal (valid in both the DOS mode and the OS/2 mode), the device driver can manipulate the linkage fields in the request packets itself by using its own linked list management.

## Memory Management

The device driver must manage addressability to data across task-time and interrupt-time operations. DevHlp services are provided to allow the device driver to be independent of the CPU mode, whether at task time or interrupt time. Addressability is particularly critical at interrupt time because the context of the current process may not cover the address space containing the data buffer that the hardware interrupt handler needs to access in order to move data.

To prevent an application from passing an unauthorized address, the device driver can use the DevHlp service VerifyAccess to validate the application's authority to access the memory. Because device drivers execute at the operating system privilege level, they have access rights to segments at all privilege levels. However, a well-balanced device driver must not allow an application to force the device driver into accessing segments which the application does not own. This check applies to addresses that an application passes within a generic IOCtl request; the OS/2 kernel validates addresses for READ and WRITE requests. If an application passes a bad address to the device driver, the device driver could halt the system if it does not verify the caller's access authority. Once an address has been verified, the device driver can proceed with the I/O request.

The DevHlp services LOCK and UNLOCK are used to fix in place a segment, which prevents the segment from being moved or swapped while the device driver needs access to it. The device driver does not need to lock segments for the READ or WRITE request packets. However, segments referenced in the generic IOCtl request packet will need to be locked by the device driver if it intends to access those segments at interrupt time.

Once a segment has been locked, the device driver can convert the virtual address (that is, the segment/selector:offset address) into a physical address with the DevHlp VirtToPhys for later use at interrupt time.

The DevHlps AllocPhys and FreePhys allow the device driver to get and free a fixed amount of memory. The device driver must use the DevHlps PhysToVirt and UnPhysToVirt to obtain a virtual address (segment/selector:offset) to access the memory.

The device driver should choose to locate critical data structures or data transfer areas in its data segment. This is best for performance when access to the structures or buffers must take place at both task time and interrupt time.

## Semaphore Management

There are two kinds of semaphores: RAM and system. The following table characterizes RAM and system semaphores:

Table 2-3. Semaphore Management

| RAM Semaphore | System Semaphore |
|---|---|
| Created by allocating a DWORD variable | Created through a dynamic link function call |
| Provides no OS/2 resource management (freeing the semaphore when the owner terminates) | Provides full OS/2 resource management (freeing the semaphore and providing notification when the owner of the semaphore terminates) |
| Used to control operations among the device driver components | Used to communicate to an application process |

RAM semaphores are defined by the semaphore user by allocating a DWORD variable and using the address in place of the handle in DevHlp semaphore services. OS/2 provides no resource management on RAM semaphores, such as releasing the semaphore when the owner terminates. System semaphores are created by an application through a dynamic link function call. OS/2 provides full resource management on system semaphores, including releasing the semaphore and notification when the owner of the semaphore terminates.

Typically, a device driver creates and uses RAM semaphores to control operations among its components. They are not restricted for use by mode. The device driver may use RAM semaphores while executing in user mode.

System semaphores are typically used by a device driver to communicate with an application process. A device driver cannot create a system semaphore, although it can use the system semaphore that the application process has created. The application process must pass the application's handle to the device driver in a generic IOCtl. The device driver then uses the DevHlp service SemHandle to obtain a semaphore handle that the device driver can use. The device driver must indicate in the SemHandle call that the system semaphore is IN-USE by the device driver. When the device driver no longer needs to use the system semaphore to communicate with the application, the device driver must call the DevHlp SemHandle and specify that the system semaphore is NOT-IN-USE.

## Character Queue Management

Character queues are used by character device drivers to buffer data. The two most frequently used structures for character buffers are the FIFO and the CIRCULAR buffer.

A character device driver may use the DevHlp services to manage a simple circular buffer for characters. The DevHlp services operate on the following character queue header:

```
CharQueue STRUC
        Qsize   DW      ?    ; Size of buffer in bytes
        Qchrout DW      ?    ; Index to next char out
        Qcount  DW      ?    ; Count of characters in buffer
        Qbase   DB      ?    ; Start of buffer
CharQueue ENDS
```

Prior to using the character queue DevHlp services, a device driver must allocate the queue header and initialize the Qsize field. The DevHlp QueueInit must be called before calling any of the other character queue DevHlps. The other fields in the queue header are managed by the character queue DevHlps and do not need to be examined or altered by the device driver.

A character device driver is not required to use the character queue DevHlp services. A character device driver can define its own character buffer management, tailored to the requirements of its buffer structure.

## Hardware Interrupt Management

The device driver's hardware interrupt handler is the component of the device driver which deals with a hardware interrupt. The hardware interrupt handler is called by the OS/2 kernel when the hardware interrupt occurs, therefore it must follow the FAR CALL/RET model. By convention, the hardware interrupt handler does not need to save and restore any registers it uses. This is done by the kernel. For performance reasons, the hardware interrupt handler will be called in the CPU mode that the hardware interrupt occurred. Therefore, the hardware interrupt handler must not have a dependency on the mode in which it is invoked.

Before the hardware interrupt handler can be invoked, its entry point must be registered for a specific hardware interrupt. This may be done during or after device driver initialization with the DevHlp service SetIRQ. Once the call to the DevHlp has been made, the hardware interrupt handler can be invoked.

Hardware interrupt sharing is not supported on the IBM Personal Computer AT™, the IBM Personal Computer XT Model 286™, or the IBM Personal System/2 Model 30 - 286™. A hardware interrupt level (IRQ) that is shared among two or more devices is referred to as a shared interrupt. Interrupt sharing is an extension of a device's design. A single interrupt level (IRQ) can be shared among two or more devices if the devices are specifically built for interrupt sharing. A device driver cannot share the hardware interrupt level without cooperation from its device. This is true for both edge-triggered and level-sensitive interrupt environments.

In an edge-triggered interrupt environment, an interrupt request will be recognized by the 8259 Programmable Interrupt Controller (PIC) as a particular edge transition (like low-to-high) on the hardware interrupt request line. The interrupt request line can remain level without generating another interrupt. The 8259 PIC requires an End-Of-Interrupt (EOI) and another of the same kind of edge transition to recognize an interrupt on that interrupt level.

In a level-sensitive interrupt environment, an interrupt request will be recognized by the 8259 PIC as a particular level on the hardware interrupt request line. The interrupt condition must be removed before the EOI is issued or else the 8259 will continue to generate interrupts for that interrupt level.

**Note:** OS/2 supports interrupt sharing only on the PS/2, which provides a level-sensitive interrupt environment, where multiple device drivers (devices) may share a particular hardware interrupt. On both the IBM Personal Computer AT and the IBM Personal Computer XT Model 286, hardware interrupts cannot be shared among multiple device drivers (devices).

The basic model for managing a hardware interrupt follows:

1. The device driver must register an interrupt handler for a hardware interrupt, specifying whether the device driver intends to share the interrupt level.
2. When invoked, the device driver interrupt handler tests the device to see if it generated the interrupt.
3. If the device has an interrupt pending or caused a spurious interrupt, the interrupt handler owns the processing of the interrupt.

   The interrupt handler services the device, resets the interrupting condition at the device, issues the End-Of-Interrupt (EOI) with the DevHlp service EOI, and RET FAR with the indicator that the interrupt handler owned the interrupt (CF = 0).
4. If the device does not have an interrupt pending, the interrupt handler does not own processing of the interrupt.

   The interrupt handler must RET FAR with the indicator that it does not own the interrupt (CF = 1).

---

™ Trademark of IBM Corporation

To permit two or more device drivers to share an interrupt level, each device driver must adhere to the rules on the following actions:

1. Interrupt Level Sharing

   All interrupt levels have the potential to be shared. There are some restrictions.

   SYSTEM TIMER RULE The system timer interrupt level (IRQ 0) cannot be shared.

   The system timer interrupt level is owned by a DOS mode interrupt handler for compatibility operations.

   ILL-BEHAVED DEVICE RULE An interrupt handler for an ill-behaved device must not share an interrupt level.

   An ill-behaved device generates interrupts before its interrupt handler is installed or cannot be told to stop generating interrupts.

   Well-behaved devices do not power-up with interrupts pending and do not remain active after their handlers have terminated. Also, well-behaved devices do not usually generate spurious interrupts.

   BIOS INT HANDLER RULE A bimodal interrupt handler uses the BIOS interrupt handler to support interrupt processing from I/O generated by the DOS mode. This bimodal interrupt handler must not share an interrupt level.

   A BIOS interrupt handler does not share its interrupt level, but assumes that it owns the interrupt processing.

2. Initializing the Interrupt Vector

   SET IRQ RULE The device driver must indicate when signing up for a hardware interrupt level with the DevHlp SetIRQ that it will share the interrupt.

   IRQ ENFORCEMENT RULE If a device driver signs up for a hardware interrupt indicating that it will not share the hardware interrupt, then a subsequent SetIRQ request to share that hardware interrupt will be refused. Conversely, if a device driver signs up for a hardware interrupt indicating that it will share the hardware interrupt, then a subsequent SetIRQ request to exclusively own the hardware interrupt will be refused.

   DOS MODE SHARING RULE Interrupt sharing cannot be performed by a DOS mode interrupt handler.

   A hardware interrupt is owned either by one or more bimodal interrupt handlers (OS/2 device drivers) or by a single DOS mode interrupt handler.

   A DOS mode interrupt handler exclusively owns its hardware interrupt. It may not share its interrupt with a bimodal device driver. A DOS mode interrupt handler may not share its interrupt because, as part of the DOS mode, it can execute only when the DOS mode is in the foreground.

   COROLLARY If a bimodal device driver (OS/2 device driver) owns a device that is accessible from a BIOS software interrupt, the bimodal device driver's interrupt handler will own the hardware interrupt level, not BIOS. However, the bimodal device driver may NOT share its interrupt level with other bimodal device drivers, if the bimodal interrupt handler uses the BIOS interrupt handler when processing an interrupt generated by an I/O request from the DOS mode.

   The bimodal device driver's interrupt handler may need to support DOS mode I/O by using the BIOS interrupt handler in the DOS mode. In this case, the bimodal device driver must be aware that the BIOS interrupt processing does not include a check for ownership of the interrupt level. In addition, the bimodal device driver must be aware of the background processing of OS/2 mode processes while the DOS mode is in the foreground. The bimodal device driver is always called in the current processor mode of operation.

   IRQ MASK RULE The operating system owns the masking of the hardware interrupt at the 8259 interrupt controller.

   The operating system will enable the hardware interrupt at the 8259 interrupt controller when the first interrupt handler signs up for the hardware interrupt. This permits the interrupt handler to communicate with its device during initialization.

3. Processing The Interrupt

STI ENTRY RULE Interrupt handlers that share interrupts will be entered with processor interrupts enabled. This prevents the lockout of higher priority hardware interrupts, because the search for the owner of the current interrupt level takes a variable amount of time.

Interrupt handlers that do not share interrupts will be entered with processor interrupts disabled.

A DOS mode interrupt handler will be entered with processor interrupts disabled for compatibility.

IRQ OWNERSHIP RULE The device driver interrupt handler, when invoked, must always interrogate its device to see if its device caused the interrupt. If the interrupt handler's device caused the interrupt, then the interrupt handler owns the processing of the interrupt.

If the interrupt handler owns the processing of the interrupt, it may briefly disable processor interrupts for critical operations. It must issue the EOI as soon as possible.

The device driver interrupt handler must be aware that once it issues the EOI, it could be reentered at its interrupt handler's entry point.

If the interrupt handler's device did not cause the interrupt, the interrupt handler must not issue an EOI.

INT RETURN RULE The interrupt handler, after taking the appropriate action in processing the interrupt, must return an indication whether it claimed the interrupt.

If the interrupt handler owns the interrupt, then it must clear the CARRY FLAG (CF = 0) and issue a FAR RET when processing is complete. If the interrupt handler does not own the interrupt, then it must set the CARRY FLAG (CF = 1) and issue a FAR RET.

SEARCH RULE The operating system calls each interrupt handler registered for a particular interrupt level until one of the interrupt handlers claims the interrupt.

EOI RULE Management of the 8259 interrupt controllers is the responsibility of the operating system. However, the End-Of-Interrupt (EOI) is the responsibility of the interrupt handler.

The interrupt handler must use the DevHlp EOI service to issue the EOI as soon as possible in the processing of its interrupt. This permits the 8259 interrupt controller to process other interrupt requests at the current interrupt priority, as well as interrupt requests of lower priorities.

In a level-sensitive interrupt environment, the EOI must not be issued to the 8259 interrupt controllers until the interrupt condition at the device is removed.

Advanced BIOS requires that all ABIOS staged-on interrupt request blocks be processed for the LID that owns the interrupt prior to the EOI. (Refer to the ABIOS EOI Placement Rule.)

PhysToVirt RULE Selectors used for PhysToVirt represent a critical resource; an interrupt handler that uses PhysToVirt must not issue the EOI until after it no longer needs the addresses generated by PhysToVirt. Otherwise the interrupt handler should disable processor interrupts before issuing the EOI; this will allow the interrupt handler to use the temporary selectors for its interrupt level without getting another interrupt on its level.

This does not apply with respect to the selectors obtained by the AllocGDT devhelp.

POSITION RULE An interrupt handler that shares an interrupt level must not depend on its position in the list of handlers for that interrupt level.

4. Advanced BIOS Considerations For Interrupt Processing

ABIOS REQUEST BLOCK RULE The interrupt handler for a particular Logical ID (LID), when invoked by the operating system, must call Advanced BIOS for each ABIOS request block that is stage-on-interrupt, even if one of the request blocks gets the return indicator that the interrupt belongs to it.

ABIOS EOI PLACEMENT RULE The EOI must be issued after all ABIOS staged-on interrupt request blocks have been processed for the LID that owns the interrupt.

ABIOS LID IRQ RULE Advanced BIOS defines only one interrupt level for each LID.

If a device driver handles more than one LID on the same interrupt level, the device driver could choose to register only one interrupt handler for any LID on that level. In this case,

the operating system will call the interrupt handler only once when the interrupt occurs. The interrupt handler must manage the processing of more than one LID in order to determine if it owns the interrupt processing for them.

In this case, the device driver interrupt handler should be aware of the Fairness Criteria problem. A LID at the end of the interrupt handler's list will not get as much service as a very active LID at the front of the list.

5. Spurious Interrupts

   In an edge-triggered interrupt environment, to handle a spurious interrupt, reset the interrupt at the 8259 interrupt controller (EOI), issue the global rearm if sharing interrupts, and enable processor interrupts. Generally, these actions would be taken by the last interrupt handler in the list of interrupt handlers.

   In a level-sensitive interrupt environment, to handle a spurious interrupt, reset the interrupt condition at the device, issue the EOI, and enable processor interrupts.

   Advanced BIOS (ABIOS) provides the capability to reset a spurious interrupt at the device through the use of an Advanced BIOS default interrupt handler for the Logical ID (LID). The interrupt handler calls the ABIOS default interrupt handler for its LID if there are no outstanding incomplete-waiting-on-interrupt request blocks. The ABIOS default interrupt handler will indicate either that the interrupt condition was successfully reset or that the interrupt did not belong to the device referenced by the LID. If the ABIOS default interrupt handler replies that the device was successfully reset, the interrupt handler must issue the EOI and return as owning the interrupt.

   Where there are incomplete-waiting-on-interrupt request blocks outstanding, ABIOS keeps track of which interrupts are expected, and will automatically service a spurious interrupt when called by the interrupt handler. The interrupt handler must call ABIOS with every request block that is incomplete-waiting-on-interrupt for a LID, even if the first one returns an indication that it performed some processing. The interrupt handler must be able to process the spurious interrupt return code from any one of these calls to the ABIOS interrupt service.

   For a device driver that directly interfaces to the device, it must check the device for the interrupt condition, even if the interrupt condition does not correspond to an outstanding I/O request. If the device had caused the spurious interrupt, the interrupt handler must reset the interrupting condition at the device, issue the EOI, and return as owning the interrupt.

   **Note:** If the device causing the spurious interrupt in the level-sensitive interrupt environment is not identified and reset, the interrupt level is locked up. This is a feature of the 8259 interrupt controller operating in level-sensitive mode.

6. DEINSTALL Considerations

   Refer to "DEINSTALL Considerations" on page 4-21 for details.

**Notes:** A single device driver with a single interrupt handler for a particular interrupt level may share its interrupt level among one or more devices that it owns. This case applies to devices of similar or same nature; for example, a printer device driver supporting more than one printer (adapter) on one interrupt level. For this case, the operating system will invoke the device driver's interrupt handler when the hardware interrupt occurs. The device driver's interrupt handler must determine which of its devices caused the interrupt; the interrupt handler will not get called for each device it manages.

If the device driver is supporting multiple devices on the same hardware interrupt, it only needs to process the first device it discovers causing an interrupt on the interrupt level in question.

Because of the level-sensitive interrupt environment, other devices that are requesting service will cause another interrupt to be generated and the device driver would be re-entered at the same point. Because of this, the device driver should strategically place the EOI to allow an orderly processing of any re-entrant interrupt requests.

However, if the device driver registers a separate unique interrupt handler entry point for each device it owns, with the interrupt handlers sharing the same interrupt level, the device driver must adhere to the interrupt sharing rules. The operating system will invoke each interrupt handler, until one handler claims the interrupt. To the operating system, each registered interrupt handler entry point appears as a separate interrupt handler. This allows the device driver's interrupt handlers to be called for each device.

# Chapter 3. Designing an OS/2 Device Driver

## Design Considerations

### Designing for Performance

For the best use of OS/2, design your device driver to conform to these standard system performance guidelines:

- Device drivers written for DOS are synchronous and poll their devices. Because DOS is a single-task operating system, a device driver may hold the CPU until I/O is complete. In the OS/2 multitasking environment, however, device drivers must surrender the CPU while they wait for I/O completion. Consequently, OS/2 device drivers must be interrupt driven.
- To provide the best performance when access to the structures or buffers must take place at both task time and interrupt time, locate critical data structures or data transfer areas in the device driver's data segment. For further details, see the section "Building a Device Driver" in this chapter.
- Design the device driver strategy routine to provide checks to yield the CPU about every 3 milliseconds (ms.). If the device driver strategy routine does not check to yield the CPU, and it executes for periods longer than 3 ms., you can delay dispatching of a process that is ready to run.
- Device drivers must be able to handle hardware interrupts without the overhead of a mode switch.

  Because the user of your device driver may have an application running in the DOS mode, hardware interrupts owned by the OS/2 device driver may occur in real mode. This means that the device driver hardware interrupt handler will be started in real mode.
- As a guideline, design both the device driver strategy routine and the hardware interrupt handler to run with processor interrupts enabled as much as possible. Plan to have the interrupt handler issue the End-Of-Interrupt (EOI) as soon as critical processing is performed.

  If your device driver does processing after sending an End-Of-Interrupt, it can receive nested interrupts. To consume as little stack space as possible, the number of nested interrupts must be limited.

  **Note:** If your device driver must support more than one level of nested interrupts, limit the number.
- Remember the time the device driver runs with interrupts disabled impacts system performance.

### Requesting OS/2 Services

Because many of the functions of an OS/2 device driver are related to system operations, OS/2 services are available using DevHlp functions. The functions are a subset of OS/2 and can operate in both the real and protect modes. Access to OS/2 services is obtained at initialization.

Some OS/2 services are limited by the context in which your device driver is running. See Chapter 5, "Device Helper Services" on page 5-1 for a complete list of the DevHlp function calls, functional descriptions, and limitations.

**Note:** Generally speaking, DevHlp services may briefly disable processor interrupts to prevent interruptions, but will preserve the entry state of the interrupt flag upon exit from the service.

## Limiting the Number of Nested Interrupts

Device driver interrupt handlers must be written to prevent uncontrolled stack growth of the interrupt stack. The interrupt handler must consume as little stack space as possible and limit the depth of nested interrupts.

The system's interrupt stack is configurable. That is, device drivers indicate their interrupt stack usage by calling the RegisterStackUsage device helper routine during initialization. This allows device drivers that require large stack space to be installed without decreasing available low memory for all users (device drivers).

When calling RegisterStackUsage, a device driver provides data describing the interrupt handler for a specified IRQ. RegisterStackUsage, therefore, is called by a device driver for each IRQ that it services. In addition to the interrupt stack size, this data includes the maximum number of interrupt levels that the device driver expects to nest. OS/2 uses this data to detect when unbounded nesting occurs and prevent it from crashing the system.

Device drivers that do processing after sending an end-of-interrupt (EOI) to the interrupt controller (8259) and enabling interrupts (STI), can receive nested interrupts. To prevent using excessive interrupt stack space, the device driver should keep internal flags to limit the amount of interrupt nesting.

The amount of interrupt nesting is limited by performing all post-EOI processing at the first-level interrupt. Nested interrupts should avoid and, if possible, eliminate post-EOI processing. In all cases, the device driver should bound the number of levels of nesting to as few as possible (preferably two), and should never permit the levels of nesting to be unbounded.

Limiting the amount of interrupt nesting to two levels, for example, can be implemented as follows:

- When a nested interrupt is encountered, a flag (or count) is set so that the post-EOI processing can be done again by the first level interrupt if necessary.
- On a first level interrupt, interrupts are enabled before doing the EOI and remain enabled during the post-EOI processing. Interrupts MUST be disabled and remain disabled before clearing the interrupt-in-progress flag and returning to the Interrupt Manager.
- On a nested interrupt, interrupts MUST be disabled and remain disabled before doing the EOI and returning to the Interrupt Manager.

**Note:** If a device driver NEEDS to support more than one level of nested interrupts, it still MUST limit the number of nested interrupts that it handles. A device driver should avoid this if at all possible.

**Note:** As device drivers allow more levels of nesting for their interrupt handlers, the potential exists for the entire interrupt stack to be consumed. This applies to all device drivers, regardless of the interrupt rate of the device being supported. It may seem that a device driver for a slow device need not follow this convention. However, if the system contains another device that has a high interrupt rate, or many devices with more moderate interrupt rates, the time between occurrence of hardware interrupt and dispatch to interrupt handler can become greater than the interrupt rate of the slow device, and excessive nesting can occur.

## Inter-Device Driver Communication

A device driver may call and pass data to another device driver by using the inter-device driver communication mechanism provided by OS/2. To communicate with each other, device drivers must maintain addressability, while being sensitive to interrupt-time performance.

A device driver indicates that another device driver may communicate with it in its device header. See the section "Device Driver Header" on page 2-11 for a detailed description of the device header.

1. The device driver must set bit 14 in the attribute field to indicate that it can participate in inter-device driver communication.
2. The device driver must set up the offset to its entry point for other device drivers, which will be identified as the IDC (inter-device driver communication) Entry Point.

Another device driver may communicate with this device driver by calling its IDC Entry Point. The other device driver gets the address of this entry point (and other information it needs to be able to call this device driver) by calling the AttachDD device helper routine. This DevHlp service is callable

during device driver initialization or at task-time kernel mode and returns the address of the IDC Entry Point of the specified 'named' device driver.

The type and form of communication are defined by the device driver that provides the IDC Entry Point. Registers may be used to indicate the type of communication (for example, initialize a buffer, write data to a buffer), as well as the form of communication (for example, data structures). To communicate with a device driver, therefore, another device driver must use the published interface to the IDC Entry Point.

## Device Monitor Support in Character Device Drivers

OS/2 provides a mechanism for applications to directly access and intercept data flowing through data streams belonging to some character device drivers. This mechanism allows applications to filter (remove, insert, or modify) data passing through a character device by registering one or more character device monitors with the device driver. The mechanism requires support by the character device driver, as well as by the application.

A *character device monitor* is an application or part of an application that uses OS/2 dynamic link function calls to interact with:

- A device driver, to gain access to its data streams by calling DosMonOpen, DosMonReg, and DosMonClose.
- A data stream, to intercept data passing through the device by calling DosMonRead and DosMonWrite.

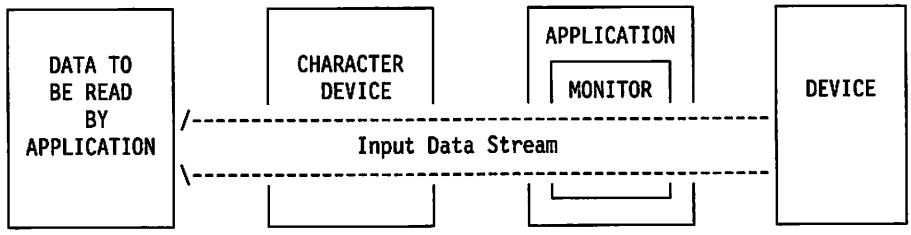



Figure 3-1. Input Device Monitor, for example, Mouse Monitor

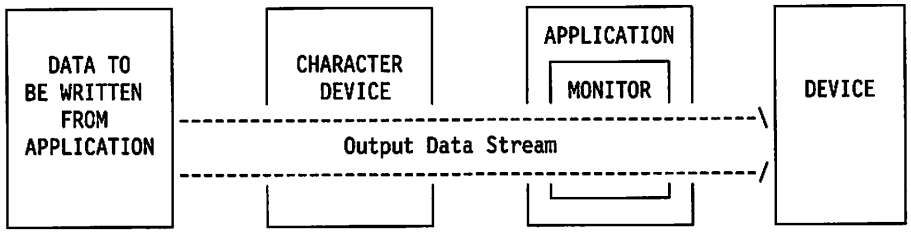



Figure 3-2. Output Device Monitor, for example, Printer Monitor

Character device monitors intercepting data from the same data stream belonging to a character device are linked together in a *monitor chain*. The first character device monitor in a monitor chain receives data directly from the device driver. This character device monitor filters the data and passes it on to the next character device monitor in the chain. This character device monitor filters the filtered data and passes it on to the next character device monitor in the chain; and so forth. The last character device monitor in the chain passes the filtered data back to the device driver.

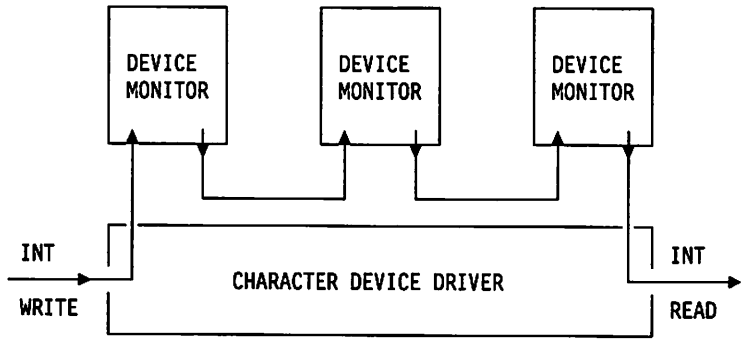



Figure 3-3. Device Monitor Chains

Character device drivers provide support for character device monitors by using the Monitor Dispatcher Device Helper routines to:

1. Create monitor chains for their data streams by calling the MonitorCreate device helper routine.
2. Register monitor buffers with their monitor chains on behalf of monitor applications by calling the Register device helper routine.
3. Send data to their monitors by calling the MonWrite device helper routine.
4. Flush all data from their monitors by calling the MonFlush device helper routine.
5. Remove monitors buffers from their monitor chains on behalf of monitor applications by calling the DeRegister device helper routine.

See Chapter 8, "Character Device Monitors" for a detailed discussion of character device monitors and code and data requirements for device monitor support in character device drivers.

## Device Support

If you are developing device drivers to be used on various system units, keep in mind that different system units may require different device drivers to run the same device. For example, the device drivers for OS/2 on IBM Personal System/2 system units are different from the device drivers for the IBM Personal Computer AT and the IBM Personal Computer XT/286. For more information refer to your hardware reference manuals, the *IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference* and Advanced BIOS supplement.

**Note:** Any device driver can be written to access hardware directly, even on a machine that has some form of BIOS interface. Writing directly to the hardware, however, makes the device driver device-dependent. Consequently, the device driver must be rewritten each time the hardware changes.

Writing to a BIOS interface that hides device-dependent features will allow a device driver to be supported across many devices.

- Consider how users of the device will make requests to the device. If users are expected to issue DosOpen, DosRead, DosWrite, and DosClose system calls, the device driver should support at least the READ and WRITE commands.
- If users issue dynamic link calls to a subsystem, and the subsystem in turn issues DosOpen, DosDevIOCtl, and DosClose calls, the device driver should support the generic IOCtl commands.

  **Note:** If a process is dependent on a device driver performing a specific operation at a specific time (operations not covered by the file I/O system), the process should use the generic IOCtl commands.
- If you want to let users erase the contents of the driver's I/O buffers, the driver should support the FLUSH INPUT and FLUSH OUTPUT commands.
- You have more freedom to decide the commands character device drivers support than block device drivers written to run under the FAT-based file system.

**Note:** This does not mean that all block device driver must run under the FAT-based file system. The FAT-based file system is the user of the block device driver and expects certain functions to be supported.

## DOS Mode Generic IOCtl Support

Following are the two types of generic IOCtls supported in the DOS mode:

- Function: AL=0DH

  This function is the same as DOS 3.3 with the addition that the register pair SI:DI is the address of the parameter block in OS/2 and DS:DX is the address of the data packet.

- Function: AL=0CH

  This function is similar to function AL=0DH except that BX contains a handle to a device instead of a drive letter. This function is useful for character devices.

The register contents are:

| *Register* | *Contents* |
|---|---|
| AH | 44H — IOCtl request |
| AL | 0DH — Drive oriented |
| | 0CH — Handle oriented |
| BL | Drive number |
| BX | Handle value |
| CH | Category |
| CL | Function |
| DS:DX | Data block |
| SI:DI | Parameter block |

Refer to the specific device IOCtl descriptions in this book for the categories and functions supported.

## DOS Mode Software Interrupt Support

The following is a list of the software interrupts supported and the compatibility exceptions for DOS mode operation:

| *Interrupt* | | *Comments* |
|---|---|---|
| 05H | Print screen | Request ignored<br>**Note:** Shift-Print screen works for text mode screens in both the OS/2 and DOS modes. |
| 12H | Memory size | Supported — size limited to DOS mode size |
| 13H | Disk / Diskette | For non-removable media only — these functions are supported:<br>01H — read status<br>02H — read sectors<br>0AH — read long<br>15H — read DASD type |
| 14H | ASYNC | If the ASYNC device driver is loaded in the system, then INT 14H will not function for its related ASYNC ports unless the utility SETCOM40 is used. See SETCOM40 in the *OS/2 Version 1.2 Command Reference* for INT 14H considerations. |
| 15H | Miscellaneous | Functions not supported:<br>87H — Block Move<br>88H — Extended memory size<br>89H — Virtual mode<br>90H — Device busy<br>91H — Int. complete<br>Functions supported with restrictions:<br>83H — Event wait<br>86H — Wait |

**Note:** Both 83H and 86H are supported; but the timer granularity is on the order of 31ms. Because the RTC (Real Time Clock) is free-running, there will be a variance of up to 1 RTC tick.

C2H — Mouse Interface

**Note:** PS/2 hardware interface not supported.

| | | |
|---|---|---|
| 17H | Printer | Supported by OS/2 device driver. |
| 19H | Reboot (Re-start) | Supported — However, this does not operate in the same manner as DOS 3.3. The system is restarted as if Ctrl-Alt-Del was pressed. |
| 1AH | TOD | Functions not supported:<br>02H — Read RTC time<br>03H — Set RTC time<br>04H — Read RTC date<br>05H — Set RTC date<br>06H — Set RTC alarm<br>07H — Reset RTC alarm |
| 1EH | Diskette parameters | Not used by device driver after boot process. |
| 24H | Hard error | Supported — OS/2 calls the application when a hard error occurs. |
| 26H | Direct write | An error is returned on requests for non-removable media. |
| 2FH | Multiplex | Returns error "Not installed, not to be installed" (printer only — request AL = 0 returns AL = 1). |
| 33H | Mouse | Supported — When the OS/2 Mouse device driver is loaded, INT 33H functions are available. |

# Using Advanced BIOS

There are two methods that device drivers may use to invoke Advanced BIOS (ABIOS): the ABIOS Transfer Convention and the Operating System Transfer Convention. For the ABIOS Transfer Convention, the ABIOS Common Entry Points are invoked to locate the specific Start, Interrupt, or Timeout entry points for the requesting Logical ID. For the Operating System Transfer Convention, the specific Start, Interrupt, or Timeout entry points must be located for the requesting Logical ID and called.

OS/2 internal device drivers use the Operating System Transfer Convention to invoke ABIOS services. User-written, installable device drivers may use either the ABIOS Transfer Convention or the Operating System Transfer Convention. DevHlps are provided for both calling conventions.

**Note:** Both kinds of device drivers mentioned above will be commonly termed ABIOS device drivers.

For performance reasons, ABIOS device drivers should call ABIOS services with processor interrupts enabled.

## Device Driver Data Segment

OS/2 recommends that an ABIOS device driver use its data segment to contain the ABIOS request blocks and data transfer buffers. The device driver data segment is located in low memory and the operating system guarantees addressability to the data segment regardless of the processor mode (protect or real). The device driver may also assume that the physical location of the device driver data segment will not move. This will allow physical data transfers to take place to buffers within the device driver's data segment.

By using its data segment, the device driver can create logical addressability to these data areas (for ABIOS) in a mode-independent manner and without interrupt disable time considerations. Physical data transfers to buffers outside of the device driver's data segment may take place if the buffer is locked.

## Obtaining a Logical ID

During its initialization, an ABIOS device driver must obtain the Logical ID (LID) for its physical device.

The allocation of a LID is managed by the operating system. This ensures that the device driver gets a unique LID for its device type. The operating system provides a DevHlp function GetLIDEntry to obtain the LID for a device driver.

The DevHlp GetLIDEntry finds the LID for the specified device ID and allocates it to the calling device driver.

The counterpart to GetLIDEntry is the DevHlp FreeLIDEntry. This service is required when the device driver DEINSTALLs or terminates to release the device driver's claim to the LID. Refer to the section "DEINSTALL Considerations" on page 4-21 for more details.

The operating system will prohibit access to a certain LID specifically the LID for System Services. The operating system management of LID access is similar to the management of hardware interrupt levels or I/O ports.

## Calling Advanced BIOS Services

For ABIOS device drivers that use the Operating System Transfer Convention, a DevHlp service (ABIOSCall) is provided to invoke ABIOS with the mode specific correct set of parameters. The device driver passes the ABIOS request block pointer, its LID, and the ABIOS primary function (start, interrupt, or timeout) to the DevHlp ABIOSCall. This sets up the stack for the call to ABIOS and calls the ABIOS function.

For ABIOS device drivers that use the ABIOS Transfer Convention, a DevHlp service (ABIOSCommonEntry) is provided to invoke the ABIOS Common Entry Points The device driver passes the mode specific pointer to the ABIOS request block and the ABIOS primary function (common start, common interrupt, or common timeout) to the DevHlp service. The DevHlp service sets up the stack and calls the requested ABIOS common entry point.

**Note:** The return code of the ABIOS function will be in the ABIOS request block.

## Mapping Device Names to LID

Having identified its LID and the number of devices or physical units the LID represents, the device driver must map each of its device names to a unit within that LID.

**Note:** A device driver supports all units under a given LID.

All device drivers are known to the operating system by device names, whether these names correspond to the ASCII string device name in the header for character device drivers or to the logical units (which correspond to drive letters) in the header for block device drivers.

In the case of a character device driver with a single device driver header, its device name must be mapped to the first unit of the LID it obtained. If the character device driver has one device header but its LID had multiple units, the rest of the units are not used.

When a character device driver has multiple device driver headers, the operating system will call the strategy routine entry point for each header during device driver initialization.

1. The first entry point called must map its device name to the first unit of the LID.
2. The second entry point called must map its device name to the next unit of the LID.
3. If the LID that was obtained by the first entry point has only one unit, the second entry point must obtain another LID and map its device name to the first unit of the second LID.
4. The device driver must start with the first LID and consume all the units before going to the next LID.

For example: The printer device driver has four device headers (PRN, LPT1, LPT2, and LPT3, respectively). The first entry point will map PRN to the first unit of the LID it obtained — let's use LID #12. If LID #12 supports only one unit, the second entry point will map LPT1 to the first unit of another LID it

must obtain — let's use LID #17. If LID #17 supports another unit, the third entry point will map LPT2 to the second unit of LID #17. If LID # 17 supports still another unit, the fourth entry point will map LPT3 to the third unit of LID #17.

In the case of a block device driver, the block device driver must obtain the necessary number of LID/units for the number of logical units it supports. The block device driver maps the first logical unit to the first-LID/first-unit, the second logical unit to the next available LID/unit, and so forth.

## Handling ABIOS Requests

Refer to "Writing a Device Driver using Advanced BIOS," for a discussion on handling requests to ABIOS.

A device driver must assume that it owns all outstanding ABIOS request blocks for a given Logical ID. During interrupt-time processing, the device driver must call ABIOS for each outstanding request that is incomplete-waiting-on-interrupt.

**Note:** This is one of the reasons that a Logical ID is not shared among device drivers.

## Writing a Device Driver using Advanced BIOS

This section describes design considerations for a device driver that uses ABIOS.

To determine the basic structure of the device driver, certain design points must be identified:

- The kind of device to be supported (character or block).
- The nature of the I/O to the device; synchronous or staged, Program I/O (PIO) or Direct Memory Access (DMA).

  A staged request can be further refined to be staged on a time delay, staged on an interrupt or both. Staged on a time delay means the operation involves waiting for a specific length of time before the operation can be continued or is completed. Staged on an interrupt means the operation involves waiting for an interrupt to occur.

  PIO or DMA refers to the type of addressing required for data transfers. PIO is done using virtual addresses (which are also referred to as logical addresses) of the form:

  ```
  segment/selector:offset.
  ```

  DMA is done using physical addresses which are 32-bit numbers, indicating the data transfer location in memory.
- The maximum number of devices.
- The maximum number of interrupt levels.

These items determine the nature of the device driver, that is, how the task-time and interrupt-time portions of the device driver relate to each other and which of the DevHlp services will be used for blocking, queueing, timers, and others.

The type and number of devices generally indicate the logical device names (for example, COM1, LPT1) that the device driver will support.

- A device type is identified by its ABIOS-architected device ID.
- A specific device is identified by a Logical ID (LID) and unit number under that LID.
- I/O to the device is performed by calling the ABIOS entry point (Start, Interrupt, or Time out) that corresponds to the particular LID.
- Parameters are passed to an ABIOS service through a request block structure.
- I/O requests can be synchronous (run to completion), or staged (run until blocked, waiting for an interrupt or time).
- Staged requests may have well-defined time delays between certain stages.
- Data transfers may use either virtual addresses of the form:

  ```
  (segment/selector:offset)
  ```

  or physical addresses.

Before using ABIOS services and during initialization, a device driver must identify every LID for which it will accept requests. To do this, the device driver uses the architected ABIOS Device ID for its device. The device driver uses the DevHlp GetLIDEntry, which searches through the ABIOS common data area looking for the LID that corresponds to the given device ID.

In general, by making repeated calls to GetLIDEntry and counting the number of units supported by each LID it obtains, the device driver determines how many supported devices are configured in the system. The device driver will only process interrupts and requests for its maximum number of supported devices. Any LID of the device driver's device type that is left over must be unclaimed so another device driver can support it.

A device driver knows which LID corresponds to a given logical device name (for example, COM1) because of the rule forcing the operating system logical device names to be in the same order as the LID entries for an associated device ID. For example, assuming one unit per LID, an installable printer device driver will support LPT3 (the third printer) by locating the third LID that corresponds to device ID of printer.

The device driver must determine which interrupt level each LID will use by using the ABIOS function, Return LID Parameters. The device driver will register interrupt handler entry points for the interrupt levels that it supports with the DevHlp SetIRQ. It keeps a list of every LID that corresponds to each interrupt handler.

**Note:** If the device driver supports multiple devices and the number of interrupt levels for those devices exceed the number of supported interrupt levels, the device driver will ignore any LID that it cannot support because too many different interrupt levels are required.

At task time, when the device driver strategy entry point for a given device header receives a request packet, the device driver knows which logical device name and LID (and unit number) correspond to that entry point.

The device driver strategy routine sets up an ABIOS request block and uses the DevHlp ABIOSCall to invoke the Advanced BIOS START routine to begin the requested ABIOS function. ABIOS requires that the ReturnCode field in the ABIOS request block be initialized to FFFFH. ABIOS will set the ReturnCode to its appropriate value.

**Note:** Either portion of the device driver, the task-time strategy routine or the interrupt handler, may start an ABIOS request. For simplicity, the example will use the strategy routine as the caller of the Advanced BIOS START service.

The pointer to the ABIOS request block and any logical data transfer pointers can be set up by the device driver independent of mode if the data transfer areas are in the device driver's data segment (segment/selector:offset).

If the data transfer is to take place to a logically addressed buffer in a bimodal environment, the device driver will need to double buffer the data transfer if the target location of the data could be high memory while running in real mode or if the device driver needs to run enabled. That is, the device driver should use an intermediate buffer located in its data segment (which is less than 1 MB) for the data transfer performed by ABIOS. The device driver would then complete the data transfer to the user data buffer itself.

In a multi-staged request, the address of the logically addressed data buffer may have to be changed by the device driver in the Advanced BIOS request block from stage to stage. This is normal because of the bimodal characteristics of the operating environment. For devices that require physical address data transfer (for example, DMA-oriented devices), the device driver must ensure that the buffer area is locked for the duration of all stages of the request.

**Interrupt during START:** If the request is staged on an interrupt then ABIOS will set the ReturnCode appropriately only when the particular service is ready to be resumed through the Advanced BIOS INTERRUPT routine. The device driver strategy routine must also set a flag to indicate whether a request has completed the START request to the point at which the strategy routine interrogates the ReturnCode. This must be done to accommodate the case where the interrupt occurs after ABIOS updates the ReturnCode, but before the device driver strategy routine interrogates the ReturnCode. In this case, the device driver interrupt handler is invoked by the interrupt and can take appropriate action on the request block, even though the device driver strategy routine has not completed processing the request block.

For example, if the strategy routine is expected to BLOCK the request until the interrupt occurs, but the interrupt handler is invoked before the strategy routine is able to BLOCK, the interrupt handler needs to flag the fact that the interrupt handler already processed the request block. The strategy routine, when it gets control, will see that it should not check the ReturnCode in the request block and it does not have to BLOCK because the request is already completed. The strategy routine will set up the request packet with the return information and return the completed request to the kernel.

This example would be more complex if, when the strategy routine got control, the request was still incomplete (as would occur in a multi-staged request). The strategy routine would still ignore the ReturnCode because the request block would already be at a different stage (than the START) but the strategy routine may still have to BLOCK.

**Interrupt after START:** For a staged ABIOS request that must wait for the interrupt associated with the specified LID to occur after the request is STARTed, the ReturnCode of the request will be set to stage-on interrupt by the ABIOS START function. This indicates that the request is incomplete. Several requests for this LID may START and be waiting for the device interrupt. These incomplete requests are commonly referred to as outstanding requests for the LID.

**Note:** The request is considered to be an outstanding request for the LID, even if the START service has not returned control to the caller of the START service.

A device driver does not assume that the return codes for an ABIOS request occur in any given order. The ReturnCode should always be checked to determine what actions to perform on the request block.

When the device driver interrupt handler is invoked by the device interrupt, it knows which LID is associated with the interrupt level. The interrupt handler must individually examine each LID associated with the interrupt level. For a LID, the interrupt handler must process all outstanding staged-on interrupt request blocks. That is, the Interrupt handler is required by ABIOS to call the Advanced BIOS INTERRUPT routine for every outstanding staged-on interrupt request block to completely process one LID. This includes a START request block in which the ReturnCode has been changed from FFFFH to stage-on interrupt but where the START service has not yet returned control to its caller. If one of the request blocks for the LID caused the interrupt, after the interrupt handler has called ABIOS with all the outstanding request blocks owned by this LID, the interrupt handler will not need to process any other LID associated with this interrupt level.

If a given LID has no outstanding ABIOS request blocks, the device driver will call the Advanced BIOS DEFAULT INTERRUPT service for that LID. The DEFAULT INTERRUPT service will reset the interrupt condition for that LID if the LID falsely caused the interrupt. It will then return to the device driver interrupt handler, indicating that the interrupt belonged to the LID.

If there is at least one outstanding ABIOS request block for a given LID, ABIOS will automatically invoke the DEFAULT INTERRUPT service if the LID generates a false interrupt. The device driver must be able to process the false interrupt return code for any call to the ABIOS INTERRUPT routine. This return code indicates that the interrupt belonged to the LID, was reset by ABIOS, and the device driver is responsible for issuing the EOI and returning to the operating system as owning the interrupt. The device driver is still responsible for calling ABIOS with any remaining staged-on-interrupt request blocks for this LID.

The device driver interrupt handler may issue the EOI (through the DevHlp EOI function) only after completely processing the LID that owns the interrupt or after the LID's DEFAULT INTERRUPT service indicates that the LID's device caused the interrupt. The Interrupt handler must process all outstanding requests under the LID that owns the interrupt, even after finding a request block which indicates that it caused the interrupt. The interrupt handler may stop processing any LID for this interrupt only when the interrupt is claimed by a LID, either by a request under the owning LID or by the owning LID's DEFAULT INTERRUPT service.

Once the interrupt handler issues the EOI after completely processing a LID, another LID requesting service at the current interrupt level would create another interrupt.

Once the EOI is issued, the interrupt handler can be reentered at its entry point. If the Interrupt handler is reentered, it must process every LID, including the one that is near completion or just completed.

In order to keep the pre-EOI processing time to a minimum, the interrupt handler may wish to issue its EOI either before it sets the return information in the operating system request packet, or before it begins processing a request packet queued by the device driver strategy routine.

If the interrupt handler did not find any LID that claimed the given interrupt either with a request block or by a DEFAULT INTERRUPT service, the interrupt handler must exit, indicating that the interrupt did not belong to it; that is, the interrupt was not caused by any LID that the device driver owns.

Eventually, the ReturnCode from ABIOS will show that the ABIOS request block is complete. The device driver can then clear the device driver request packet queue and take the next request packet and try to start another ABIOS request block.

The device driver will support the time-out requirements of ABIOS with the DevHlp SetTimer and the DevHlp TickCount. Instead of counting every clock tick, the device driver will use TickCount to force its timer tick entry point to receive control as infrequently as possible. The design of the ABIOS TIMEOUT function is in one-second increments.

Device drivers in an ABIOS environment have the same requirement to support DOS mode operations as they do in OS/2. In addition, for certain devices such as diskette and disk, the device driver must reset the device when switching between ABIOS operation and BIOS operation. The state of the device information is kept internally to the ABIOS and BIOS device blocks, which would get out of synchronization if the device were not reset when switching between the two sets of code.

## Using DevHlp Services for Address Conversion

There are some basic guidelines when using the DevHlp services PhysToVirt and UnPhysToVirt for address conversion.

- Use ES:DI whenever possible when converting a single physical address.
- Use ES:DI for the first address conversion when using two physical addresses.
- Check the physical address pair and convert the physical address above 1 MB first.

The following examples are recommended ways of using these DevHlps in various situations. These examples apply to both task-time and interrupt-time operations, except where noted.

- To get a logical address to place in an ABIOS request block:
  1. Call PhysToVirt with ES:DI for the converted address.
  2. Store the converted address in the ABIOS request block.
  3. Call the ABIOS service.
  4. Call UnPhysToVirt.
- To convert a single physical address to use as the source in a data transfer to a logical address, (that is, one that was passed as input for this data transfer request):
  1. Save DS.
  2. Call PhysToVirt with DS:SI for the converted address.
  3. Perform the data transfer.
  4. Restore DS.
  5. Call UnPhysToVirt.
- To provide two logical addresses in order to do a data transfer:
  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.
  4. Call PhysToVirt with DS:SI for the second address.
  5. Perform the data transfer.
  6. Restore DS.

7. Call UnPhysToVirt.

- To do multiple data transfers:
  1. Examine the first physical address pair. If one of the physical addresses is above 1 MB, convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.
  4. Call PhysToVirt with DS:SI for the second address.
  5. Perform the data transfer.
  6. Restore DS.
  7. Examine the second physical address pair. If one of the physical addresses is above 1 MB, convert it first.
  8. Call PhysToVirt with ES:DI for the first address.
  9. Save DS.
  10. Call PhysToVirt with DS:SI for the second address.
  11. Perform the data transfer.
  12. Restore DS.
  13. Perform these steps until all data transfers are complete.
  14. Call UnPhysToVirt.
- To provide two logical addresses for a data transfer which must be broken down into smaller chunks in order to YIELD periodically:
  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.
  4. Call PhysToVirt with DS:SI for the second address.
  5. Perform the data transfer on the chunk.
  6. Restore DS.
  7. Call UnPhysToVirt.
  8. YIELD.
  9. When control is returned, repeat these steps.
- To pass two logical addresses to a subroutine, one of which must be converted from a physical address, the other is obtained from the device driver's data segment:
  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, convert it first.
  2. Call PhysToVirt with ES:DI for the address to convert.
  3. Save the converted address in the appropriate input parameter to the subroutine.
  4. Save the other logical address (located in the device driver's data segment) in the appropriate input parameter to the subroutine.
  5. Call the subroutine.
  6. Call UnPhysToVirt.
- To use two logical addresses to do a data transfer at interrupt time before the EOI:
  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.

4. Call PhysToVirt with DS:SI for the second address.
5. Perform the data transfer.
6. Restore DS.
7. Call UnPhysToVirt.
8. Issue the EOI.

- To use two logical addresses in order to do a data transfer at interrupt time after the EOI:
  1. Issue the EOI.
  2. Examine the physical address pair. If one of the physical addresses is above 1 MB, convert it first.
  3. Save the interrupt flag.
  4. Disable interrupts.
  5. Call PhysToVirt with ES:DI for the first address.
  6. Save DS.
  7. Call PhysToVirt with DS:SI for the second address.
  8. Perform the data transfer.
  9. Restore DS.
  10. Restore the interrupt flag.
  11. Call UnPhysToVirt.

# Building a Device Driver

Writing your own device driver allows new or optional devices to be installed without modifying the operating system.

To create a device driver that OS/2 can install:

1. *Create your DATA and CODE segments.*

   A device driver may have multiple code and data segments. (See "Device Driver Program Model" on page 2-10.) The first segment after the .EXE file header must be the first data segment. (See Figure 2-3 on page 2-10.) The first data segment must originate at 0, not at 100H.

   Because a device driver may have multiple code and data segments, it may make FAR calls to other code segments. The device driver must examine the code segment value and set the appropriate value for the CPU mode in the CS register when it calls another one of its code segments. Similarly, before accessing another one of its data segments, the device driver must examine the data segment value and set the appropriate value for the CPU mode in the DS register.

   **Note:** Because OS/2 installs the driver anywhere in memory, care must be taken in any memory references. Do not expect that your driver will be loaded at the same place every time.

   By default, only the first data and code segments remain in memory after initialization of the device driver. All other segments are assumed to be initialization code, unless otherwise designated. Memory used by these segments is returned to the system.

2. *Define your device header.*

   The device header must be the first data in the first data segment. The device header must be located immediately after the .EXE file header.

   The device header defines the characteristics of a device driver for OS/2. See "Device Driver Header" on page 2-11 for a detailed description of each device header field.

3. Define a strategy routine and interrupt handler routine in your code segments.

4. Define your code and data segments and their attributes in the linker .DEF file (module definition file).

   The SEGMENTS directive and its IOPL option in the .DEF file indicate which segments are to be permanent, that is, remain in memory after initialization.

5. Link the code and data segments as a library.
6. Include a DEVICE= statement for the device driver in your CONFIG.SYS file so that your device driver is installed when the system is initialized.
7. Prepare a Device Driver Profile (see Figure 2-1 on page 2-3.) to include on a device support diskette so that users may install your device driver by using the DDINSTAL utility.

## Device Driver Sample Program (DEMODD.ASM)

The device driver sample program demonstrates the following:

- How a device driver's strategy routine and timer handler are structured.
- How queued requests are handled by the timer handler.
- How a character device driver provides character device monitor support.

See Chapter 8, "Character Device Monitors" for detailed information on monitor support for character device drivers.

The device driver sample program is included on the Sample Programs Diskette in the *OS/2 Version 1.2 Programmer's Toolkit.*

# Chapter 4. Device Driver Strategy Commands

## Request Packets

The device driver strategy routine is called with ES:BX pointing to the request packet. The pointer to the request packet (ES:BX) is bimodal. That is, the pointer is valid in both the DOS mode and the OS/2 mode.

OS/2 does not guarantee that the order of API requests issued by multiple threads will be preserved in the order that the corresponding request packets arrive at the device driver. Multiple application threads, or threads created due to DosReadAsync and DosWriteAsync, can get blocked in the operating system. This allows a device driver request packet (for an API request by a subsequent thread that does not get blocked) to arrive out of order. A device driver is responsible for providing a synchronization mechanism between itself and application processes, if it supports multiple outstanding requests. Also, request packet ordering must be preserved.

A request packet consists of two parts, the request header and the command-specific data field. The structure of the request packet is detailed in Table 4-1.

Table 4-1. Request Packet structure

| Field | Length |
|---|---|
| Length of Request Packet | BYTE |
| Block Device Unit Code | BYTE |
| Command Code | BYTE |
| Request Packet Status | WORD |
| Reserved | DWORD |
| Queue Linkage | DWORD |
| Command-specific Data | BYTES |

where:

**Length of Request Packet**
is set to the total length in bytes of the request packet (the length of the request header plus the length of the command-specific data).

**Block Device Unit Code**
identifies the unit for which the request is intended. This field has no meaning for character devices.

**Command Code**
indicates the requested device driver function. The device driver command codes are summarized in Table 4-4 on page 4-4.

**Request Packet Status**
is defined only for Open and Close request packets on entry to the strategy routine. For all other request packets, the status field is undefined on entry.

For an Open request packet, Bit 3 (08H) of the status field is SET if the packet was generated from a DosMonOpen; otherwise it was a DosOpen.

For a Close request packet, Bit 3 (08H) of the status field is SET if the packet was generated by a DosMonClose or a DosClose of a handle that was generated by a DosMonOpen (So that monitor handles generated that are left open when a process exits will be closed properly). Otherwise, it was a DosClose on a non-monitor handle.

On exit from the strategy routine the status field describes the resulting state of the request as shown in Table 4-2.

| Table 4-2. Request Packet Status Field | | | | | |
|---|---|---|---|---|---|
| **15** | **14** | **13-10** | **9** | **8** | **7-0** |
| ERROR | DEV ERROR | RESERVED | BUSY | DONE | ERROR CODE (bit 15 on) |

Bit 15 is the Error bit. If this bit is set, the low 8 bits of the status word (7-0) indicate the error code. The error code is processed by OS/2 in one of the following ways:

1. If the IOCTL category is 'User Defined', reference the Category Code under Generic IOCTL Commands, FF00H is ORed with the byte-wide error code;
2. If not 'User Defined' and Bit 14 (device driver defined error code) is set, FE00H is ORed with the byte-wide error code;
3. Otherwise, the error code must be one of those shown in Table 4-3 below and is mapped into one of the standard OS/2 API return codes.

Bit 14 is a device driver defined error if set in conjunction with bit 15.

Bits 13 - 10 are reserved.

Bit 9 is the Busy bit. It is only set by status calls and the removable media call. See "STATUS" and "REMOVABLE MEDIA" in this chapter for more information about the calls.

Bit 8 is the Done bit. If it is set, it means the operation is complete. The driver sets the done bit to 1 when it exits.

Bits 7-0 are the low 8 bits of the status word. If bit 15 is set, bits 7-0 contain the error code. The error codes and errors are shown in Table 4-3.

| Table 4-3 (Page 1 of 2). Status Field Error Codes | |
|---|---|
| **Error Codes** | **Description** |
| 00H | Write protect violation |
| 01H | Unknown unit |
| 02H | Device not ready |
| 03H | Unknown command |
| 04H | CRC error |
| 05H | Bad drive request structure length |
| 06H | Seek error |
| 07H | Unknown media |
| 08H | Sector not found |
| 09H | Printer out of paper |
| 0AH | Write fault |
| 0BH | Read fault |
| 0CH | General failure |
| 0DH | Change disk (logical switch) |
| 0EH | Reserved |

| Table 4-3 (Page 2 of 2). Status Field Error Codes | |
|---|---|
| **Error Codes** | **Description** |
| 0FH | Reserved |
| 10H | Uncertain media |
| 11H | Character I/O call interrupted |
| 12H | Monitors not supported |
| 13H | Invalid parameter |

Uncertain media (10H) should be returned when the state of the media in the drive is uncertain. This response should NOT be returned to the INIT command. For fixed disks, the device driver must begin in a media uncertain state in order to have the media correctly labelled. In general, the following guidelines may be used to determine when to respond with uncertain media:

- When a drive-not-ready condition is detected. (In this case, return uncertain media to all subsequent commands until a reset media command is received.)
- When accessing removable media without change-line support, and a time delay of two or more seconds has occurred.
- When the state of the change-line indicates that the media may have changed.

Character I/O call interrupted (11H) should be returned when the thread performing the I/O was interrupted out of a DevHlp Block before completing the requested operation.

Monitors not supported (12H) should be returned for monitor commands (monitor open/close, register IOCtl), if monitors are not supported by the device driver.

Invalid parameter (13H) should be returned when one or more fields of the request packet contain invalid values.

**Queue Linkage Field**
is provided to maintain a linked list of request packets. The device driver may use the request queue management DevHlp services, or it may use its own queue management.

**Note:** Because a pointer to a request packet is bimodal (valid in both the DOS mode and the OS/2 mode), the pointer may be used directly in the queue linkage rather than a 32-bit physical address.

**Command-Specific Data**
are the parameters required for the device driver command. The commands and the actual formats of the corresponding request packets are discussed in the following sections.

# Summary of Device Driver Commands

Table 4-4 is a summary of device driver strategy commands.

Table 4-4. Device Driver Commands

| Code | Function | Block | Char |
|---|---|---|---|
| 0H | INIT | * | * |
| 1H | MEDIA CHECK | * | |
| 2H | BUILD BPB | * | |
| 3H | Reserved | | |
| 4H | READ (input) | * | * |
| 5H | NONDESTRUCTIVE READ NO WAIT | | * |
| 6H | INPUT STATUS | | * |
| 7H | INPUT FLUSH | | * |
| 8H | WRITE (output) | * | * |
| 9H | WRITE WITH VERIFY | * | * |
| AH | OUTPUT STATUS | | * |
| BH | OUTPUT FLUSH | | * |
| CH | Reserved | | |
| DH | DEVICE OPEN | * | * |
| EH | DEVICE CLOSE | * | * |
| FH | REMOVABLE MEDIA | * | |
| 10H | GENERIC IOCtl | * | * |
| 11H | RESET MEDIA | * | |
| 12H | GET LOGICAL DRIVE MAP | * | |
| 13H | SET LOGICAL DRIVE MAP | * | |
| 14H | DEINSTALL | | * |
| 15H | PORTACCESS | | * |
| 16H | PARTITIONABLE FIXED DISKS | * | |
| 17H | GET FIXED DISK/LOGICAL UNIT MAP | * | |
| 18H | Reserved | | |
| 19H | Reserved | | |
| 1AH | Reserved | | |
| 1BH | Reserved | | |

The commands are described in detail in the following section of this chapter.

**Note:** All DWORD pointers are stored with offset first, then segment.

Initialize the device.

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| Data_1 | BYTE |
| Pointer_1 | DWORD |
| Pointer_2 | DWORD |
| Data_2 | BYTE |

## Remarks

On entry, the request block contains the following fields as inputs to the device driver:

Pointer_1 Points to the DevHlp Entry Point

Pointer_2 Points to the INIT arguments

Data_2 Drive number for the first block device unit

The DevHlp Entry Point is a bimodal address and is valid in both the DOS mode and the OS/2 mode. The DevHlp Entry Point is called to invoke a service specified in the DL register.

The arguments for installable device drivers from the DEVICE= line in the CONFIG.SYS file allow the device driver to use configurable parameters to initialize itself and its device.

At initialization time, the device driver runs as a thread under a protect mode process at application level with I/O privilege. The device driver may issue certain OS/2 dynalink function calls at this time. Refer to "Device Driver Initialization" on page 2-7 for more details.

On completion of initialization, the device driver must set fields in the request packet as described:

***Field*** ***OUTPUT Information for INIT success***

Data_1 Number of logical block devices or units (block devices only)

Pointer_1 WORD offset to end of code segment

WORD offset to end of data segment

Pointer_2 Points to the BIOS Parameter Block (BPB) array for the logical block devices or units (block devices only)

Status Set the status word in the request header to 0100H

A block device driver must return in Data_1 the number of logical devices or units that are available. The kernel's file system layer will assign sequential drive letters to these units. A character device driver will set Data_1 to 0.

Both block device drivers and character device drivers must set Pointer_1 with the offsets of the code and data segments. This allows a device driver to release code and data needed only by the device driver's initialization routine. First, the initialization code and data must be located at the end of the appropriate segments. Then, as the final step in initialization, the device driver sets the offsets to the end of the code segment and the end of the data segment. This also permits a device driver to load with a maximum-sized data segment (64 KB) and let it release the amount that it does not need.

# 0H / INIT
# Initialize Device Driver

**Note:** Remember that the device driver code and data segments reside in memory below 640K. The DOS mode requires contiguous memory below 640K. Although memory returned by the device driver from its data segment is available to the system, it is not available for the DOS mode.

A block device driver must return an array of BPBs for the logical units that it supports in Pointer_2. A character device driver will set Pointer_2 to 0.

The Status field in the request packet header must be set to indicate no error and done.

If the device driver determines that it cannot set up the device and wants to quit, it should return with the error bit in the request packet status field set to 1. The device driver can also return the following:

| ***Field*** | ***OUTPUT Information for INIT failure*** |
|---|---|
| Data_1 | BYTE 00H |
| Pointer_1 | WORD 0000H |
| | WORD 0000H |
| Status | 810CH |

The status field in the request packet header must be set to indicate the failure of the INIT request with the General Failure error return code. The Status must also indicate that the request is done.

One of the above techniques must be used to return device initialization failures from the device driver to the system initialization process.

A character device driver that contains multiple device driver headers can fail initialization on a subset of the headers in its header chain.

The system initialization process remembers the last non-zero size code and data segment offsets returned for the devices in the device driver that completed initialization. These last values are used to re-size the device driver's code and data segments after INIT packets have been sent to the device driver for each device in the device driver header chain.

When a device in the header chain cannot be initialized, the device driver can set the code and data segments to 0, and/or set the error bit in the request packet status field to indicate initialization failure for that device. The device driver will not receive any future request packets for a specific device if it returns a failure for the INIT request packet for that device. If none of the devices in the device driver header chain pass initialization, the device driver will not remain loaded.

Because the system initialization process maintains the pass/fail return status for each device header in a device driver header chain, the device driver should not manipulate the linkages of the headers.

Determine the state of the media.

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| Media descriptor | BYTE |
| Return code | BYTE |
| Return pointer to previous volume ID if supported | DWORD |

## Remarks

On entry, the request packet will have the media descriptor field set for the drive identified in the request packet header.

The device driver must perform the following actions for the MEDIA CHECK request:

- Set the status word in the request packet header.
- Set the return code where:

  -1 = Media has been changed
  0 = Unsure if media has been changed
  1 = Media unchanged

Examples of DOS media descriptor bytes:

Table 4-5. Examples of DOS media descriptor bytes

| Disk Type | # Sides | # Sectors/Track | Media Descriptor |
|---|---|---|---|
| Fixed disk | -- | -- | F8H |
| 3 1/2-inch | 2 | 9 | F9H |
| 3 1/2-inch | 2 | 18 | F0H |
| 5 1/4-inch | 2 | 15 | F9H |
| 5 1/4-inch | 1 | 9 | FCH |
| 5 1/4-inch | 2 | 9 | FDH |
| 5 1/4-inch | 1 | 8 | FEH |
| 5 1/4-inch | 2 | 8 | FFH |
| 8-inch | 1 | 26 | FEH |
| 8-inch | 2 | 26 | FDH |
| 8-inch | 2 | 8 | FEH |

**Note:** To determine whether you are using a single-sided or a double-sided 8-inch diskette (FEH), attempt to read the second side. If an error occurs you can assume the diskette is single-sided.

For 8-inch diskettes:

FEH (IBM 3740 Format). Single-sided, single density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 1 reserved sector, 2 File Allocation Tables (FATs), 68 directory entries, 77*26 sectors.

FDH (IBM 3740 Format). Double-sided, single density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 4 reserved sectors, 2 FATs, 68 directory entries, 77*26*2 sectors.

FEH Double-sided, double density, 1024 bytes per sector, soft sectored, 1 sector per allocation unit, 1 reserved sector, 2 FATs, 192 directory entries, 77*8*2 sectors.

Application programmers are encouraged to use the generic IOCtl — Get Device Parameters (Category 8, function 63) and reference the BPB (BIOS Parameter Block) to determine the type of media.

Build the BIOS Parameter Block (BPB). This is requested when the media has changed or when the media type is uncertain.

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| Media descriptor | BYTE |
| Transfer address | DWORD |
| Pointer to BPB table | DWORD |
| Drive number | BYTE |

## Remarks

On entry, the request packet will have the media descriptor set for the drive identified in the request packet header. The transfer address is a virtual address to a buffer containing the boot sector media if the block device driver attribute field has bit 13 set; otherwise the buffer contains the first sector of the File Allocation Table (FAT).

The device driver must perform the following actions:

- Set the pointer to the BPB table
- Update the media descriptor
- Set the status word in the request header.

The device driver must determine the media type in the drive in order to return the pointer to the BPB table. Previously, the FAT ID byte determined the structure and layout of the media. Because the FAT ID byte has only eight possible values (F8 through FF), it is clear that, as new media types are invented, the available values will soon be exhausted. With the varying media layouts, OS/2 needs to be aware of the location of the FATs and directories before it reads them.

The device driver reads the boot sector from the specified buffer. If the boot sector is for DOS 2.00, 2.10, 3.10, 3.20, or OS/2, the device driver returns the BPB from the boot sector. If the boot sector is for DOS 1.00 or 1.10, the device driver reads the first sector of the FAT into the specified buffer. The FAT ID is examined and the corresponding BPB is returned. Only two formats are possible for diskettes formatted by a 1.00 or 1.10 system, 5 1/4-inch single-sided (FEH) and 5 1/4-inch double-sided (FFH.)

# 2H / BUILD BPB
# Build BIOS Parameter Block

The information relating to the BPB for a particular media is kept in the boot sector for the media.

## Boot Sector Format

Table 4-6. Boot Sector Format

| Field | Length |
|---|---|
| Short JUMP (EBH) followed by a NOP (90H) | 2 BYTES |
| OEM name and version | 8 BYTES |
| Bytes per sector | WORD |
| Sectors per allocation unit (must be a power of 2) | BYTE |
| Reserved sectors (starting at logical sector 0) | WORD |
| Number of FATs | BYTE |
| Number of root directory entries (maximum allowed) | WORD |
| Number of sectors in logical image (total sectors in media, including boot sector, directories, for example.) | WORD |
| Media descriptor | BYTE |
| Number of sectors occupied by a single FAT | WORD |
| Sectors per track | WORD |
| Number of heads | WORD |
| Number of hidden sectors | WORD |

The last three WORDs above help the device driver understand the media. The number of heads is useful for supporting different multiple head drives that have the same storage capacity but a different number of surfaces. The number of hidden sectors is useful for supporting drive partitioning schemes.

For drivers that support volume identification and disk change, this call should cause a new volume identification to be read off the disk. This call indicates that the disk was properly changed.

Read from or write to a device.
Read From (4H) / Write To (8H) / Write with Verify (9H)

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| Media descriptor | BYTE |
| Transfer address | DWORD |
| Byte / sector count | WORD |
| Starting sector number for block device | DWORD |
| System File Number | WORD |

## Remarks

On entry, the request packet will have the media descriptor set for the drive identified in the request packet header. The transfer address is a 32-bit physical address of the buffer for the data. The byte/sector count is set to the number of bytes to transfer (for character device drivers) or the number of sectors to transfer (for block device drivers). The starting sector number is set for the block device drivers. The System File Number is a unique number associated with an open request.

The device driver must perform the following actions:

- Perform the requested function.
- Set the actual number of sectors or bytes transferred.
- Set the status word in the request header.

The DWORD transfer address in the request packet is a locked 32-bit physical address. The device driver can pass it to the DevHlp function PhysToVirt to obtain a segment swapping address for the current mode. The device driver does not need to unlock the address when the request is completed.

**Note:** The functions IOCtl READ and IOCtl WRITE are not supported by the OS/2 device drivers.

# 5H / NONDESTRUCTIVE READ NO WAIT
# Nondestructive Input

Read character from buffer but do not remove it.

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| Returned character | BYTE |

## Remarks

The device driver must perform the following actions:

- Return a byte from the device
- Set the status word in the request header.

For input on character devices with a buffer, the device driver returns from this function with the busy bit set to 0 along with a copy of the first character in the buffer. The busy bit is set to 1 to indicate no characters in the buffer. This function allows the operating system to look ahead one input character without blocking in the device driver.

Determine input or output status on character devices.
Input Status (6H) / Output Status (AH)

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |

## Remarks

The device driver must perform the following actions:

- Perform the requested function.
- Set the busy bit.
- Set the status word in the request header.

For output on character devices, if the busy bit is returned set to 1, a subsequent write request to the device driver would have to wait for the completion of a currently active request. If the busy bit is returned set to 0, there is no current request. Therefore, a write request would start immediately.

For input on character devices with a buffer, if the busy bit is returned set to 1, there are no characters currently buffered in the device driver. If the busy bit is returned set to 0, there is at least one character in the device driver buffer. The effect of busy bit = 0 is that a read of one character will not need blocking. Devices that do not have an input buffer in the device driver should always return busy = 0.

# 7H, BH / FLUSH
# Input or Output Flush

Flush or terminate all pending requests.
Input Flush (7H) / Output Flush (BH)

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |

## Remarks

The device driver must perform the following actions:

- Perform the requested function.
- Set the status word in the request header.

This call tells the device driver to flush (terminate) all known pending requests. Its primary use is to flush the input (or output) queue on character devices.

Open or close the device.
Open Device (DH) / Close Device (EH)

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| System File Number | WORD |

## Remarks

The System File Number is a unique number associated with an open request.

The device driver must perform the following actions:

- Perform the requested function.
- Set the status word in the request header.

Character device drivers may use OPEN/CLOSE requests to correlate using their devices with application activity. For instance, the device driver may increase a reference count for every OPEN and decrease the reference count for every CLOSE. When the count goes to 0, the device driver can flush its buffers. This can be thought of as a "last close causes flush," or as the device driver using the OPEN as an indicator to send an initialization string to its device.

For example, to ensure that a printer is in a known state at the start of an I/O stream, this call could be used to set the font and page size. Similarly, the CLOSE call can be used to send a post-string (like a form feed) at the end of an I/O stream. Using IOCtl to set these pre-strings and post-strings provides a flexible mechanism of serial I/O device stream control.

# FH / REMOVABLE MEDIA
# Check for Removable Media

Check for removable media.

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |

## Remarks

The device driver must perform the following actions:

- Set the busy bit of the status word.

  Set the busy bit to 1 if the media is non-removable.
  Set the busy bit to 0 if the media is removable.

- Set the status word in the request header.

The device driver receives this request packet when an application issues an IOCtl function call to determine whether it is dealing with a removable or non-removable media drive. For example, removable or non-removable drives may print different versions of some prompts.

Send I/O control commands to a device.

## Format of Request Block

text = (DosDevIOCtl).

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| Function category | BYTE |
| Function code | BYTE |
| Parameter Buffer Address | DWORD |
| Data Buffer Address | DWORD |
| System File Number | WORD |

## Format of Request Block

text = (DosDevIOCtl2).

| Field | Length |
|---|---|
| Request header | 13 BYTES |
| Function category | BYTE |
| Function code | BYTE |
| Parameter Buffer Address | DWORD |
| Data Buffer Address | DWORD |
| System File Number | WORD |
| Parameter Buffer Length | WORD |
| Data Buffer Length | WORD |

## Remarks

On entry the request packet will have the IOCtl category code and function code set. The parameter buffer and the data buffer addresses will be set as virtual addresses. Note that some IOCtl functions do not require data and/or parameters to be passed. For these IOCtls the parameter and data buffer addresses may contain zeros. The System File Number is a unique number associated with an open request.

If the device driver indicates in the function level in the device attribute field of its device header that it supports DosDevIOCtl2, the Generic IOCtl request packets passed to the device driver will have two additional words containing the lengths of the Parameter Buffer and Data Buffer, respectively. If the device driver indicates through the function level that it supports DosDevIOCtl2, but the appliction issues DosDevIOCtl, the Parameter Buffer and Data Buffer length fields will be set to zero.

The device driver must perform the following actions:

- Perform the requested function.
- Set the status word in the request header.

The device driver is responsible for locking the parameter and data buffer segments, and converting the pointers to 32-bit physical addresses, if necessary.

# 10H / GENERIC IOCtl
# I/O Control for Devices

Refer to the *OS/2 Version 1.2 Programming Reference* and the *OS/2 Version 1.2 Programming Guide* for more detailed information on the generic IOCtl interface for applications (DosDevIOCtl). Refer to Chapter 7, "Generic IOCtl Commands" in this book for a detailed description of the generic IOCtl interface for device drivers.

Reset the Uncertain Media error condition and allow OS/2 to identify the media.

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |

## Remarks

On entry, the unit code identifies the drive number to be reset.

The device driver must perform the following actions:

- Set the status word in the request header.
- Reset the error condition for the drive.

Before this command, the device driver had returned the error "Uncertain Media" for the drive. This action informs the device driver that it no longer needs to return the error for the drive.

# 12H, 13H / LOGICAL DRIVE
# Get/Set Logical Drive Mapping

Get/Set which logical drive is currently mapped onto a particular unit.
Get Logical Drive Mapping (12H) / Set Logical Drive Mapping (13H)

## Format of Request Block

| Field | Length |
|---|---|
| Request header | 13 BYTES |

## Remarks

On entry, the unit code contains the unit number of the drive for which this operation is to be performed.

The device driver must perform the following actions:

- For Get, it must return the logical drive that is mapped onto the physical drive indicated by the unit number in the request header.
- For Set, it must map the logical drive represented by the unit number onto the physical drive that has the mapping of logical drives.
- The logical drive is returned in the unit code field. This field is set to 0 if there is only one logical drive mapped onto the physical drive.
- Set the status word in the request header.

Terminate the character device driver.

## Format of Request Block

| Field | Length |
|---|---|
| Request Header | 13 BYTES |

## Remarks

When a device driver is loaded, the attribute field and name in its header are used to determine if the new device driver is attempting to replace a driver (device) already installed. If so, the previously installed device driver is requested by the operating system to DEINSTALL the indicated device. If the installed device driver refuses the DEINSTALL command, the new device driver is not allowed to initialize. If the installed device driver performs the DEINSTALL, the new device driver is initialized.

If the character device driver honors the DEINSTALL request, it must perform the following actions:

- Release any allocated physical memory.
- UnSet any hardware vectors that it had claimed.
- JMP to the previous handler to preserve the DOS mode vector chain (it cannot reset the vector), if the device driver has a software interrupt handler.
- Perform any other cleanup.
- Clear the error bit in the status field to indicate a successful DEINSTALL.

If the character device driver determines that it cannot or will not abort, it should:

- Set the error bit in the status field and set the error code to 03H, UNKNOWN COMMAND.

### DEINSTALL Considerations

**Logical IDs:** An ABIOS device driver maps its device name to a unit within a Logical ID (LID). It receives a DEINSTALL request for its device name, which implies a single unit of a LID. To honor the DEINSTALL request, it must relinquish the LID by way of the DevHlp FreeLIDEntry at DEINSTALL time.

**Note:** To release a LID means to release all units under that LID. For a LID that has multiple units, the device driver must discontinue support of all units under the LID. If multiple units correspond with multiple device headers in the device driver data segment, the device driver must note which device header corresponds to each unit in the DEINSTALL LID, and discontinue support.

**Hardware Interrupts:** In honoring a DEINSTALL command, a device driver must remove its claim on the interrupt level. The DevHlp UnSetIRQ provides this service.

If the device driver's device is ill-behaved (that is, it cannot be told to stop generating interrupts or be quiesced), the device driver must not remove its interrupt handler. In this case, the device driver must refuse the DEINSTALL request.

**Note:** Because of the general interrupt sharing capabilities in a level-sensitive interrupt environment, device drivers should not assume that the DevHlp SetIRQ service can be used to determine whether a given device is being used by another device driver. Instead, the DEINSTALL convention should be used on the logical device name that another device driver may be using to access the same device.

# 16H / PARTITIONABLE FIXED DISKS
# General query of device support

This call is used by the system to ask the device driver how many physical-partitionable fixed disks the device driver supports.

## Format of Request Block

| Field | Length |
|---|---|
| Request Header | 13 BYTES |
| Count | BYTE |
| Reserved | WORD |
| Reserved | WORD |

## Remarks

This is done to allow the Category 9 Generic IOCtls to be routed appropriately to the correct device driver. This call is not tied to a particular unit that the device driver owns, but is directed to the device driver as a general query of its device support.

The device driver must perform the following actions:

- Set the count as discussed above (1-based).
- Set the status word in the request header.

This call is used by the system to determine which logical units supported by the device driver exist on physical partitionable fixed disk N.

## Format of Request Block

| Field | Length |
|---|---|
| Request Header | 13 BYTES |
| Units-supported bit mask | 4 BYTES |
| Reserved | WORD |
| Reserved | WORD |

## Remarks

On entry the request packet header unit field identifies a physical disk number (based on 0) instead of a logical unit number. The device driver returns a bit map of which logical units exist on the physical drive. The physical drive relates to the partitionable fixed disks reported to the system by way of the PARTITIONABLE FIXED DISKS command. It is possible that no logical units exist on a given physical disk because it has not yet been initialized.

The device driver must perform the following actions:

- Set the 4-byte bit mask to indicate which logical units that it owns exist on the physical partitionable fixed disk for which the information is being requested.
- Set the status in the request packet header.

The bit mask is set up as follows: A 0 means the logical unit does not exist and a 1 means it does. The first logical unit that the device driver supports is the low-order bit of the first BYTE. The bits are used from right to left starting at the low order bit of each following BYTE. It is possible that all the bits will be 0.

For example; a block device driver supports five units spread over the two diskette drives and one partitionable fixed disk in a system. Unit 0 and unit 1 map to the diskette drives. Unit 2, 3, and 4 map to the fixed disk. For the device command, this device driver will set the 4-byte bit map to:

```
'0000 0000 0000 0000 0000 0000 0001 1100' binary
```

or

```
'00 00 00 1C' hex
```

# Chapter 5. Device Helper Services

Many of the functions of an OS/2 device driver are related to system operations rather than to hardware operations. An interface to operating system services is available to device drivers through the *DevHlp interface*.

Access to these system services is obtained at the time of device driver initialization. The request packet for the INIT command contains a pointer to the DevHlp interface. The pointer to the DevHlp interface is a bimodal pointer; that is, this pointer to the DevHlp interface is valid in both the DOS mode and the OS/2 mode. The device driver does not have to be sensitive to the mode of operation before requesting DevHlp services.

The DevHlp services are listed below. A service is invoked by setting up the appropriate registers, loading a function code into the DL register, and making a FAR CALL to the DevHlp interface routine, whose address was supplied at device driver initialization time.

## DevHlp Services and Function Codes

| DevHlp Service | Code | Description |
|---|---|---|
| SchedClockAddr | 0 | Get system clock routine |
| DevDone | 1 | Device I/O complete |
| Yield | 2 | Yield the CPU |
| TCYield | 3 | Yield the CPU to time-critical |
| Block | 4 | Block thread on event |
| Run | 5 | Unblock thread |
| SemRequest | 6 | Claim a semaphore |
| SemClear | 7 | Release a semaphore |
| SemHandle | 8 | Get a semaphore handle |
| PushReqPacket | 9 | Add request to list |
| PullReqPacket | A | Remove request from list |
| PullParticular | B | Remove a specific request from list |
| SortReqPacket | C | Insert request in sorted order to list |
| AllocReqPacket | D | Get a request packet |
| FreeReqPacket | E | Free request packet |
| QueueInit | F | Initialize character queue |
| QueueFlush | 10 | Clear character queue |
| QueueWrite | 11 | Put a character in the queue |
| QueueRead | 12 | Get a character from the queue |
| Lock | 13 | Lock segment |
| Unlock | 14 | Unlock segment |
| PhysToVirt | 15 | Map physical-to-virtual address |
| VirtToPhys | 16 | Map virtual-to-physical address |
| PhysToUVirt | 17 | Map physical-to-user virtual |
| AllocPhys | 18 | Allocate physical memory |
| FreePhys | 19 | Free physical memory |
| SetROMVector | 1A | Set Software interrupt vector |
| SetIRQ | 1B | Set a hardware interrupt handler |
| UnSetIRQ | 1C | Reset a hardware interrupt handler |
| SetTimer | 1D | Set a timer handler |
| ResetTimer | 1E | Remove a timer handler |
| MonitorCreate | 1F | Create a monitor |
| Register | 20 | Install a monitor |
| DeRegister | 21 | Remove a monitor |
| MonWrite | 22 | Pass data records to monitor |
| MonFlush | 23 | Remove all data from stream |
| GetDOSVar | 24 | Return pointer to DOS variable |
| SendEvent | 25 | Indicate an event |
| ROMCritSection | 26 | ROM BIOS critical section |
| VerifyAccess | 27 | Verify memory access |
| Reserved | 28 | |

| DevHlp Service | Code | Description |
|---|---|---|
| Reserved | 29 | |
| AttachDD | 2A | Attach to a device driver |
| InternalError | 2B | Signal an internal error |
| Reserved | 2C | |
| AllocGDTSelector | 2D | Allocate GDT Descriptors |
| PhysToGDTSelector | 2E | Map physical to virtual address in GDT |
| RealToProt | 2F | Switch from Real Mode to Protect Mode |
| ProtToReal | 30 | Switch from Protect Mode to Real Mode |
| EOI | 31 | Issue an End-Of-Interrupt |
| UnPhysToVirt | 32 | Mark PhysToVirt complete |
| TickCount | 33 | Modify timer |
| GetLIDEntry | 34 | Get Logical ID |
| FreeLIDEntry | 35 | Release Logical ID |
| ABIOSCall | 36 | Invoke ABIOS function |
| ABIOSCommonEntry | 37 | Invoke ABIOS Common Entry Point |
| RegisterStackUsage | 38 | Indicate Stack Usage |

## DevHlp Services and Device Contexts

As discussed in the section on device driver architecture, device driver code may run in one of four contexts.

**Kernel mode -** the context in which the device driver strategy routine runs.

**Interrupt mode -** the context in which the device driver hardware interrupt handler runs.

**User mode -** the context in which the device driver handler for a DOS mode software interrupt runs.

**Init mode -** the context in which the device driver Strategy routine runs when it is called with the INIT request packet.

Certain restrictions apply as to when individual DevHlp services can be used. The following table Table 5-1, outlines which DevHlp services may be called in each device context.

Table 5-1 (Page 1 of 3). DevHlp Service Modes

| DevHlp Service | Code | Kernel | Inter-rupt | User | Init |
|---|---|---|---|---|---|
| SchedClockAddr | 0 | * | | | * |
| DevDone | 1 | * | * | | |
| Yield | 2 | * | | | |
| TCYield | 3 | * | | | |
| Block | 4 | * | | * | |
| Run | 5 | * | * | * | |
| SemRequest | 6 | * | | * | |
| SemClear | 7 | * | * | * | |
| SemHandle | 8 | * | * | | |
| PushReqPacket | 9 | * | | | |
| PullReqPacket | A | * | * | | |
| PullParticular | B | * | * | | |
| SortReqPacket | C | * | | | |
| AllocReqPacket | D | * | | | |
| FreeReqPacket | E | * | | | |

| Table 5-1 (Page 2 of 3). DevHlp Service Modes | | | | | |
|---|---|---|---|---|---|
| **DevHlp Service** | **Code** | **Kernel** | **Inter-rupt** | **User** | **Init** |
| QueueInit | F | * | * | * | * |
| QueueFlush | 10 | * | * | * | |
| QueueWrite | 11 | * | * | * | |
| QueueRead | 12 | * | * | * | |
| Lock | 13 | * | | | * |
| Unlock | 14 | * | | | * |
| PhysToVirt | 15 | * | * | | * |
| VirtToPhys | 16 | * | | | * |
| PhysToUVirt | 17 | * | | | * |
| AllocPhys | 18 | * | | | * |
| FreePhys | 19 | * | | | * |
| SetROMVector | 1A | * | | | * |
| SetIRQ | 1B | * | | | * |
| UnSetIRQ | 1C | * | * | | * |
| SetTimer | 1D | * | | | * |
| ResetTimer | 1E | * | * | | * |
| MonitorCreate | 1F | * | | | * |
| Register | 20 | * | | | |
| DeRegister | 21 | * | | | |
| MonWrite | 22 | * | * | * | |
| MonFlush | 23 | * | | | |
| GetDOSVar | 24 | * | | | * |
| SendEvent | 25 | * | * | | |
| ROMCritSection | 26 | | | * | |
| VerifyAccess | 27 | * | | | |
| AttachDD | 2A | * | | | .* |
| InternalError | 2B | * | * | * | * |
| AllocGDTSelector | 2D | | | | * |
| PhysToGDTSelector | 2E | * | * | * | * |
| RealToProt | 2F | * | * | | |
| ProtToReal | 30 | * | * | | |
| EOI | 31 | | * | | * |
| UnPhysToVirt | 32 | * | * | | * |
| TickCount | 33 | * | * | * | * |
| GetLIDEntry | 34 | * | | * | * |
| FreeLIDEntry | 35 | * | | * | * |
| ABIOSCall | 36 | * | * | * | * |
| ABIOSCommonEntry | 37 | * | * | * | * |

| Table 5-1 (Page 3 of 3). DevHlp Service Modes | | | | | |
|---|---|---|---|---|---|
| **DevHlp Service** | **Code** | **Kernel** | **Inter-rupt** | **User** | **Init** |
| RegisterStackUsage | 38 | | | | * |

Related DevHlp services are grouped together in the following categories.

1. System Clock Management
   - SchedClockAddr
2. Process Management
   - Block
   - DevDone
   - Run
   - TCYield
   - Yield
3. Semaphore Management
   - SemClear
   - SemHandle
   - SemRequest
4. Request Queue Management
   - AllocReqPacket
   - FreeReqPacket
   - PullParticular
   - PullReqPacket
   - PushReqPacket
   - SortReqPacket
5. Character Queue Management
   - QueueFlush
   - QueueInit
   - QueueRead
   - QueueWrite
6. Memory Management
   - AllocGDTSelector
   - AllocPhys
   - FreePhys
   - Lock
   - PhysToGDTSelector
   - PhysToUVirt
   - PhysToVirt
   - Unlock
   - UnPhysToVirt
   - VerifyAccess
   - VirtToPhys
7. Interrupt Management
   - EOI
   - SetIRQ
   - SetROMVector
   - UnSetIRQ
8. Timer Services
   - ResetTimer
   - SetTimer
   - TickCount
9. Monitor Management

- DeRegister
- MonFlush
- MonitorCreate
- MonWrite
- Register

10. System Services
    - GetDOSVar
    - SendEvent
    - ROMCritSection

11. Processor Mode Services
    - ProtToReal
    - RealToProt
12. Advanced BIOS Services
    - ABIOSCall
    - ABIOSCommonEntry
    - FreeLIDEntry
    - GetLIDEntry

## DevHlp Interfaces

Calling conventions for and functional descriptions of each DevHlp service are described in the following sections of this chapter. In addition to the explicit effects noted under each service, the interrupt flag can be set or cleared by some services, and other flags can be affected by the calls. Some services require that the interrupt flag be off when they are called.

The device driver can assume that the state of the interrupt flag will be preserved, and that the DevHlp routine will not enable interrupts unless stated otherwise in the functional description for each routine. The only exceptions apply to functions that allow the device driver to relinquish control of the CPU. Therefore, during calls to functions such as Yield and TCYield, the device driver cannot assume that interrupts will remain disabled.

All registers except flag registers are preserved across DevHlp calls unless specified as containing return parameters.

This routine is used to invoke an ABIOS service for the Operating System Transfer Convention.

## Processing

```
MOV  AX,LID              ;Logical ID
MOV  SI,RB_Offset        ;Offset in data segment
                         ; to ABIOS request block
MOV  DH,Entry_Point      ;Specifies entry point
                         ; 0 = start
                         ; 1 = interrupt
                         ; 2 = timeout
MOV  DL,DevHlp_ABIOSCall
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
    ABIOS service invoked.
'C' Set if error
    AX = error code
         ABIOS not present.
       Unknown ABIOS Command
```

## Remarks

The stack is set, depending on the current address mode, for the call to ABIOS, and the indicated ABIOS function is called according to the Operating System Transfer Convention. Refer to Operating System Transfer Convention on page 3-6. When the ABIOS function returns, ABIOSCall will clean up the stack before returning to the device driver.

**Note:** Advanced BIOS functions called in user mode may need to be protected from being suspended in the background. This will occur when the DOS mode is in the foreground and the user selects an OS/2 mode application to run. The DOS mode will be suspended. To protect the ABIOS function, the device driver should issue the DevHlp call ROMCritSection.

Note that DS must point to the device driver's data segment. If DS had been previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

# ABIOSCommonEntry
# Invoke ABIOS Common Entry Point

This service is used to invoke an ABIOS Common Entry Point according to the Advanced BIOS Transfer Convention.

## Processing

```
MOV  SI,RB_Offset          ;Offset in data segment
                           ; to ABIOS request block
MOV  DH,Entry_Point        ;Specifies entry point
                           ; 0 = start
                           ; 1 = interrupt
                           ; 2 = timeout
MOV  DL,DevHlp_ABIOSCommonEntry
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
    ABIOS service invoked.
'C' Set if error
    AX = error code
         ABIOS not present.
         Unknown ABIOS Command
```

## Remarks

ABIOSCommonEntry sets up the stack, depending on the current address mode, for the call to one of the Advanced BIOS Common Entry Points. It then invokes the indicated ABIOS common entry point. On return from the ABIOS function, the ABIOSCommonEntry cleans up the stack before returning to the device driver.

**Note:** Advanced BIOS functions called in user mode may need to be protected from being suspended in the background. This will occur when the DOS mode is in the foreground and the user selects an OS/2 mode application to run. The DOS mode will be suspended. To protect the ABIOS function, the device driver should issue the DevHlp call ROMCritSection.

Note that DS must point to the device driver's data segment. If DS had been previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

This function allocates a set of GDT selectors for a device driver to use. This allocation is performed at device driver INIT time.

## Processing

```
MOV  ES,address_high          ; 32-bit address of GDT
MOV  DI,address_low           ;   selector array
MOV  CX,number                ;Number of selectors requested
MOV  DL,DevHlp_AllocGDTSelector
CALL [Device_Help]
```

## Results

```
'C' Cleared if successful
'C' Set if error
    AX = error code
         Invalid address
         Zero selectors requested
         Not enough selectors available
```

## Remarks

AllocGDTSelector is used to allocate a set of GDT selectors for a device driver to use for bimodal task-time and interrupt-time operations.

The address passed in ES:DI gives the location of an array of words to be filled in with the GDT selectors allocated. The value of CX specifies the number of selectors to be allocated. Note that the selector values returned may not be contiguous values.

A bimodal device driver supports both real mode and protect mode I/O and must be able to transfer data without being dependent on the current mode of operations. In addition, the interrupt handler of a bimodal device driver must be able to address data buffers regardless of the context of the current process (the current LDT will not necessarily address the data space that contains the data buffer that the interrupt handler needs to access). The PhysToGDTSelector function is used to establish the addressability of a GDT selector, and the GDT selector's addressability will remain valid and the same until another PhysToGDTSelector call is made for the same selector.

The RealToProt function is used to change processor mode from real mode to protect mode, and the ProtToReal function is used to return the processor to real mode. The device driver must always return the processor to the same mode it was in when entered if either the ProtToReal or RealToProt functions are used.

# AllocPhys
# Allocate Fixed Block of Physical Memory

AllocPhys is used by device drivers to allocate a block of fixed memory.

## Processing

```
MOV  BX,size_low            ;size in bytes
MOV  AX,size_high
MOV  DH,high_or_low         ;Relative position to 1 megabyte
                            ; = 0  above 1 megabyte
                            ; = 1  below 1 megabyte
MOV  DL,DevHlp_AllocPhys
CALL [Device_Help]
```

## Results

```
'C' Clear if memory allocated
    AX:BX = 32-bit physical address.
'C' Set if memory not allocated
    AX = error, memory not allocated.
```

## Remarks

The memory allocated by this function is fixed memory, and may not be "unfixed" through the Unlock call.

If memory is requested to be allocated high (above 1 megabyte), and no memory above 1 megabyte is available, then an error is returned. The device driver could then attempt to allocate low memory.

Conversely, if memory is requested to be allocated low (below 1 megabyte), and no memory below 1 megabyte is available, then an error is returned and the device driver could try allocating high memory, if appropriate.

This service returns a pointer to an empty request packet.

## Processing

```
MOV  DH,wait_flag              ;Wait for available
                               ; request packet
                               ; = 0 if to wait
                               ; = 1 if not to wait
MOV  DL,DevHlp_AllocReqPacket
CALL [Device_Help]
```

## Results

```
'C' Cleared if a request packet was allocated.
    ES:BX  is the virtual address of
      the allocated request packet.
'C' Set if a request packet was not allocated.
```

## Remarks

AllocReqPacket returns a bimodal pointer to a maximum size request packet. The bimodal pointer is a virtual address that is valid for both the DOS mode and the OS/2 mode.

Some device drivers, notably the disk device driver, need to have additional request packets to service task-time requests. Because device drivers are bimodal, they cannot use a packet residing in their data segment because the resulting pointer is not bimodal.

Request packets that were allocated by an AllocReqPacket may be placed in the request packet linked list.

Request packets allocated in this manner should be returned to the kernel as soon as possible through the FreeReqPacket function. The system as a whole has a limited number of request packets, so it is important that a device driver not allocate request packets and hold them for future use.

The state of the interrupt flag is not preserved across calls to this DevHlp.

# AttachDD
# Attach to a Device Driver

This function returns the address of the Inter-Device Driver Communication (IDC) Entry Point to a specified device.

## Processing

```
MOV BX,OFFSET DS:ddname          ; Name of other device driver
MOV DI,OFFSET DS:dd@             ; Data area to hold address
MOV  DL,DevHlp_AttachDD
CALL [Device_Help]
```

## Results

```
'C' Clear if no error. Data area filled in.
'C' Set if error.
    Device driver not found or
    no IDC entry point.
```

## Remarks

The ddname field contains the ASCII name of the target device driver. If the target device driver is a character device driver, the ddname must match the name in the target device driver's device header.

The dd@ field must be 12 bytes in length. It has the following layout:

| Description | Length |
|---|---|
| Real Mode OFFSET of IDC Entry Point | WORD |
| Real Mode CS SEGMENT of IDC Entry Point | WORD |
| Real Mode DS of IDC Device Driver | WORD |
| Protect Mode OFFSET of IDC Entry Point | WORD |
| Protect Mode CS SELECTOR of IDC Entry Point | WORD |
| Protect Mode DS of IDC device driver | WORD |

Before the device driver calls the entry point, it must verify that the entry point it received is non-zero for the CPU mode it wants to call the IDC entry point.

When calling the IDC entry point of the target device driver, the caller must set the DS register for the target device driver. Other registers used in the calling convention must be defined by the target device driver.

The IDC entry point of the target device driver must follow the FAR CALL/RET model.

This service "sleeps" the thread executing in the device driver until either the Run call is issued on the event identifier or a timeout occurs.

## Processing

```
MOV  BX,event_id_low          ;Low word of event id
MOV  AX,event_id_high         ;High word of event id
MOV  DI,time_limit_high       ;Timeout interval
MOV  CX,time_limit_low        ;in milliseconds
                              ; = -1 if to never timeout
MOV  DH,interruptible_flag    ;Tells if sleep is interruptible
                              ; = 0  interruptible
                              ; = 1  non-interruptible
MOV  DL,DevHlp_Block
CALL [Device_Help]
```

## Results

```
'C' Clear if event wakeup.
'C' Set if unusual wakeup.
'Z' Set if wakeup due to timeout.
'Z' Cleared if sleep was interrupted.
AL  Awake code, non-zero if unusual wakeup.
    Interrupts enabled.
```

## Remarks

The return from the Block call indicates whether the "wakeup" occurred as the result of a RUN call or an expiration of the time limit.

Block removes the current thread from the run queue and starts executing some other thread. The thread blocked in the device driver is reactivated and Block returns when Run is called with the same event identifier, when the time limit expires, or when the thread is signalled.

The event identifier is an arbitrary 32-bit value, but a convention must be followed to coordinate with the thread issuing the Run function. The standard convention for Block/Run operations is to use the address of some structure or memory cell associated with the reason for blocking and running. For example, a thread blocking until some resource is cleared normally blocks "on" the address of the ownership flag for that resource.

Because dual-mode device drivers may be Blocked in one mode and Run in the other, using the virtual address as the event identifier is not sufficient; the virtual address of an item in one mode is not the same as its virtual address in the other mode. A possible option for a device driver is that it Blocks/Runs on the 32-bit address of a request packet being processed for the thread.

The Block/Run mechanism is so designed that it cannot guarantee immunity from a faulty wakeup by some other thread in the system running a 32-bit key value, which happens to match some unrelated blocking thread's key. The goal is to choose keys which can be known to the Blocker and the Runner and have a high likelihood of being unique. Users of Block/Run must always check the reason for their wakeup to make sure that the event really took place and that the wakeup was not accidental.

When calling Block it is important to use the sequence:

```
Disable Interrupts
while (need to wait)
      Block (value)
      Disable Interrupts
```

Interrupts are turned off *before* checking the condition (I/O done, resource freed, whatever) first to avoid a deadlock by getting an interrupt-time Run call before completing the call to Block. Block re-enables the interrupts. Also note the "while clause". It is essential to recheck the awaited condition and, if necessary, re-disable interrupts and re-call Block. The convention of using an address as an event identifier should prevent double use of an identifier.

## Block
## This Thread From Running

A time limit of -1 means that Block waits indefinitely until Run is called. Block can only be called by the task-time portion of a device driver.

When using Block to block a thread, the driver can specify whether or not the sleep may be interrupted. If the sleep is interruptible, then the kernel can abort the blocked thread and return from the Block without using a corresponding Run. In general, the sleep should be marked as interruptible unless the sleep duration is expected to be short, that is, less than a second.

If the return from the Block indicates that the sleep was interrupted, that means that some internal event occurred that requires attention (such as a signal, process death, or some other forced action). The device driver should respond by performing any necessary cleanup, setting the error code in the status field of the request packet, setting the done bit, and returning the request packet to the kernel.

DeRegister removes all of the monitors associated with the specified task from the specified monitor chain.

## Processing

```
MOV  BX,monitor_PID          ; Process ID of monitor task
MOV  AX,monitor_handle       ; MonitorCreate handle for chain
MOV  DL,DevHlp_DeRegister
CALL [Device_Help]
```

## Results

```
'C' Clear if OK
    AX = # of monitors still registered in chain,
         after Deregistration.
'C' Set if error
    AX = error code
         Invalid monitor handle
         Invalid parameter
```

## Remarks

This function may only be called at task time in the OS/2 mode. If this function is called in the DOS mode, an invalid parameter error is returned to the device driver.

To remove a monitor from a monitor chain, the device driver must supply the PID of the process that owns the monitor being removed and the handle of the monitor chain that is affected. All monitors belonging to the specified PID are removed from the monitor chain.

A single process may register more than one monitor with the same monitor chain. Therefore, DeRegister removes all monitors associated with the specified process from the specified monitor chain.

See Chapter 8, "Character Device Monitors" for more information on the use of this call.

# DevDone
# Flag I/O Complete

This function signifies that a request has completed and unblocks any threads waiting in the kernel for the request.

## Processing

```
LES  BX,request_packet       ;Pointer to I/O request
                             ; packet.
MOV  DL,DevHlp_DevDone
CALL [Device_Help]
```

## Results

None

## Remarks

This service will set the "done" bit in the status field of the request packet header and issue RUNs on threads which are blocked in the kernel waiting for the request packet to be completed. DevDone is called from the device interrupt routine. The device driver should set any error flags in the status field before calling the routine.

The virtual address of a request packet is bimodal. (The virtual address is valid in both the DOS mode and the OS/2 mode.) Therefore, the device driver may pass the request packet pointer to DevDone without being sensitive to the mode of operations.

DevDone does not apply to request packets that were allocated from the AllocReqPacket function call.

The device driver does not have to call DevDone for requests that are completed at task time (in the strategy routine). Requests that are completed at strategy time should return with the done bit set in the request packet.

This routine is used to issue an End-Of-Interrupt to the master/slave 8259 interrupt controllers as appropriate to the interrupt level.

## Processing

```
MOV  AL,IRQnum              ;Interrupt level number
                            ; ( 0 - F )
MOV  DL,DevHlp_EOI
CALL [Device_Help]
```

## Results

None

## Remarks

This routine is used to issue an End-Of-Interrupt to the 8259 interrupt controllers on behalf of a device driver interrupt handler. If the specified interrupt level is for the slave 8259 interrupt controller, then this routine will issue the EOI to both the master and slave 8259s.

Device drivers must use this service in their interrupt handlers for upward compatibility.

**Note:** This routine is callable at Init-time for interrupt processing.

This DevHlp does not change the state of the interrupt flag. If the device driver returns to the operating system immediately after issuing the EOI, then it should disable interrupts prior to the EOI. Disabling interrupts prior to issuing the EOI allows the processing for this interrupt level to be completed before the system services the next interrupt. This reduces the probability of excessive nested interrupts causing a system stack overflow.

If any post EOI work is done by the INT handler, it should be limited to the first or non-nested interrupt. Nested INT processing should be done only prior to the EOI.

# FreeLIDEntry
# Release a Logical ID

This routine is used to release a Logical ID. This must be done at DEINSTALL or termination time.

## Processing

```
MOV  AX,LID                 ;Logical ID from
                            ; GetLIDEntry
MOV  DL,DevHlp_FreeLIDEntry
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
'C' Set if error
    AX = Error Code
         Not your LID.
         LID does not exist.
         ABIOS not present.
```

## Remarks

The attempt to free a Logical ID may fail if the device driver does not own the LID or if the LID does not exist.

**Note:** DS must point to the device driver's data segment. If DS was previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

FreePhys is used to release memory allocated by the AllocPhys call.

## Processing

```
MOV  BX,address_low          ;32-bit physical address
MOV  AX,address_high         ;
MOV  DL,DevHlp_FreePhys
CALL [Device_Help]
```

## Results

```
'C' Cleared if memory freed
'C' Set if memory not freed
    Cannot free memory not allocated with AllocPhys.
```

## Remarks

Any memory that the device driver allocated by way of the AllocPhys should be released prior to device driver termination.

# FreeReqPacket
# Free an Allocated Request Packet

This service is used to release a request packet previously allocated by AllocReqPacket

## Processing

```
LES  BX,request_packet       ;Pointer to request packet
                             ; previously allocated.
MOV  DL,DevHlp_FreeReqPacket
CALL [Device_Help]
```

## Results

None

## Remarks

The device driver should only free a request packet that had been previously allocated by AllocReqPacket. The DevDone service should not be used to return an allocated request packet.

The system as a whole has a limited number of request packets, so it is important that a device driver not allocate request packets and hold them for future use.

The state of the interrupt flag is not preserved across calls to this DevHlp.

# This routine is used to return the address of a kernel variable.

GetDOSVar returns the address of a kernel variable.

## Processing

```
MOV  AL,varnumber           ;Variable wanted.
MOV  DL,DevHlp_GetDOSVar
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error
    AX:BX points to the variable.
'C' Set if error
```

## Remarks

The following is the list of *read only* variables:

| ***Index*** | ***Description of variable*** |
|---|---|
| 1 | SysINFOseg:WORD<br><br>Bimodal segment address of the System Global InfoSeg. Valid at both task time and interrupt time. |
| 2 | LocINFOseg:DWORD<br><br>Selector/Segment address of the local (LDT) INFO segment. Valid only at task time. |
| 3 | Reserved |
| 4 | VectorSDF:DWORD<br><br>Pointer to the stand-alone dump facility. Valid at both task time and interrupt time. |
| 5 | VectorReboot:DWORD<br><br>Pointer to restart the OS/2. Valid at both task time and interrupt time. |
| 6 | Reserved<br><br>Pointers to the MSATS facility, OS/2 mode and DOS mode. Valid at both task time and interrupt time. |
| 7 | YieldFlag:BYTE<br><br>Indicator for performing yields. Valid only at task time. |
| 8 | TCYieldFlag:BYTE<br><br>Indicator for performing time-critical yields. Valid only at task time. |
| 9 | Reserved |
| 10 | Reserved |
| 11 | DOS mode Code Page Tag Pointer: DWORD Segment/offset of the current code page tag of DOS mode. Valid only at task time. |

These variables are maintained by the kernel for the benefit of device drivers.

The returned pointer is a bimodal pointer, that is, the returned address is valid in either the DOS mode or the OS/2 mode. Note that the address returned is the "address" of the indicated variable; the variable may contain a vector to some facility or it may contain some structure.

# GetLIDEntry
# Get a Logical ID

This routine is used to obtain a Logical ID (LID) for devices that exist (that is, devices that are "awake").

## Processing

```
MOV  AL,DeviceID            ;Desired Device ID
MOV  BL,RelativeLID#        ;Nth Logical ID  of this Device ID
                            ; (0 - FF) where 0 = first unclaimed
                            ; LID (that is, first one available)
                            ; 1 = the first LID
MOV  DH,DeviceState         ;Requested LID indicator
                            ; 0 = all other LIDs
                            ; 1 = DMA, POS
MOV  DL,DevHlp_GetLIDEntry
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
    AX = LID number
'C' Set if error
    AX = error code
         LID already owned.
         LID does not exist.
         ABIOS not present.
```

## Remarks

GetLIDEntry is used by a device driver to obtain a LID entry. Because OS/2 does not support the Advanced BIOS Sleep/Wake functions, only devices that are "awake" are considered to exist and thus available to device drivers.

This function may be employed in two ways. One way is for the device driver to specify a relative LID. Because the ordering of LIDs corresponds to ordering of the physical devices, a device driver that desires to support a certain relative device can determine if a LID entry is available. (An example is a character device driver that supports COM4; that is, it wishes to get the LID entry for the fourth COM port.)

The other way to use this function is for the device driver to request the first available LID for its device type. (An example is a block device driver that wishes to get the first available LID for diskettes.)

In either use of this function, GetLIDEntry will search the ABIOS Common Data Area table for an entry corresponding to the specified Device ID. If an entry is located that matches the caller's form of request, it is returned to the caller. If a LID entry is found but already owned, an error is returned. If no LID entry is found, an error is also returned.

Some LIDs will not be allocated to device drivers. Certain Device IDs are used by the operating system kernel to perform such actions as mode switching. The reserved Device IDs are:

- System Services.

Certain LIDs will be allocated as shared. For these Device IDs, GetLIDEntry will allow multiple device drivers to access the LID concurrently. It is up to the device driver to determine if the device is busy or available for use when needed. The list of Device IDs that are allocated as shared follows:

- DMA
- POS.

**Note:** GetLIDEntry must be called with the DeviceState parameter set to 1 in order to obtain a LID for these Device IDs. In all other cases, DeviceState must be set to 0.

**Note:** DS must point to the device driver's data segment. If DS was previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

This function is called when an internal inconsistency has been detected.

## Processing

```
LDS  SI, message_text
MOV  DI, message_text_length
MOV  DL, DevHlp_InternalError
CALL [Device_Help]
```

## Results

This routine **never** returns to the caller.

## Remarks

This DevHlp routine should be used only when an internal inconsistency is detected. In this situation, the system might be unable to continue safely. This should only be used when the operation of the system is questionable. A minor device driver (printer, com, or others) might never use this routine.

The maximum message length is 128 characters. Longer messages are truncated.

The message display code is limited. The message should contain ASCII characters in the range 0x20-0x7E, with optional 0x0D and 0x0A to break the text into multiple lines.

The message text must be prefaced with the device driver name.

# Lock
# Lock Memory Segment

Lock is called by device drivers at task time to lock a memory segment. If the segment is unavailable, the caller must specify whether Lock should block until the segment is available or should return immediately. A long term lock returns, as a result of a fail, even if the caller specified the request block until available.

## Processing

```
MOV  AX,segment_@          ;Selector or segment
MOV  BH,typeflag           ;Type of lock
                           ;  Bit 0 (duration):
                           ;        = 0 if short-term
                           ;        = 1 if long-term
                           ;  Bit 1 (where)
                           ;        = 0 if any memory
                           ;        = 1 if high memory
                           ;  Bit 2 (type)
                           ;        = 0 if normal lock
                           ;        = 1 if verify lock
MOV  BL,waitflag           ;= 0 Block till available
                           ;= 1 Return if it is not
                           ;    immediately available
MOV  DL,DevHlp_Lock
CALL [Device_Help]
```

## Results

```
'C' Clear if segment locked
    AX:BX  = lock handle
'C' Set if segment unavailable or invalid handle.
    AX, BX not preserved
```

## Remarks

Only the following combinations of type flags are valid:

00h = short-term, any memory

Segment is marked fixed at its current physical address.

01h = long-term, any memory

Segment is marked fixed, and the system may move it into the region reserved for fixed segments. Thus, the physical address of the segment may be changed by the Lock call.

03h = long-term, high memory

Segment is marked fixed, and the system may move it into the region reserved for fixed segments. Thus, the physical address of the segment may be changed by the Lock call. If the Lock succeeds, the segment is guaranteed to be in high memory. This type is only available at initialization time. No lock handle is returned, and the selector must be that of a segment from the device driver's load image. This lock is irreversible.

04h = short term, any memory, verify lock

Segment is not made present, and remains swappable. It will not be freed or shrunk until the verify lock is removed. You may block when accessing a segment that has been verify locked, but valid access is guaranteed.

A long term lock will return, as a result of a fail, even if the caller specified the request "block until available."

**Note:** The duration of the lock must be set to LONG-TERM, or a verify lock should be used, for operations that are expected to take 2 seconds or more to complete. Use of short-term locks for longer periods of time can prevent an adequate amount of movable, swappable memory from being available for system use, thus halting the system. A device driver may use short-term locks where necessary for its own code and data segments, provided the 2-second rule is not violated.

**Note:** The Lock call need only be done on segments that have not been locked already by the OS/2 kernel (for example, an address that is passed to the device driver through an IOCTL).

**Note:** If the device driver plans to Lock the segment down, it should call the Lock DevHlp before calling the VerifyAccess DevHlp.

# MonFlush
# Flush Data from Monitor Chain

MonFlush removes all data from the specified monitor chain (such as the data stream).

## Processing

```
MOV  AX,monitor_handle       ;MonitorCreate handle for chain
MOV  DL,DevHlp_MonFlush
CALL [Device_Help]
```

## Results

```
'C' Clear if OK
    AX = 0
'C' Set if error
    AX = error code
         Invalid monitor handle
         Invalid parameters
```

## Remarks

This function may be called at task time in the OS/2 mode only. If this function is called in the DOS mode, an invalid parameter error is returned to the device driver.

When a device driver calls MonFlush, the OS/2 Monitor Dispatcher creates and places a *flush record* into the monitor chain. The general format of monitor records requires that every record contains a flag word as the first entry. One of the flags is used to indicate that this record is a flush record. The flush record consists of only the flag word. This record is used by monitors along the chain to reset internal state information and to assure that all internal buffers are flushed. The flush record must be passed along to the next monitor because the monitor dispatcher will not process any more information until the flush record is received at the end of the monitor chain, that is, is returned to the device driver's monitor chain buffer at the end of the monitor chain

**Note:** Subsequent MonWrite requests will fail (or block) until the flush completes, that is, until the flush record is returned to the device driver's monitor chain buffer.

The state of the interrupt flag is not preserved across calls to this DevHlp.

See Chapter 8, "Character Device Monitors" for more information on the use of this call.

MonitorCreate creates an initially empty chain of monitors or removes an empty chain of monitors.

## Processing

```
LES  SI,final_buffer     ;Address of device driver's
                                 ; monitor chain buffer
LDS  DI,notify_rtn       ;Address of notification routine
MOV  AX,Handle           ;Handle for this chain
                         ;  = 0  create new monitor chain
                         ; < > 0  specifies chain to be removed
                                  (returned from previous create call)
MOV  DL,DevHlp_MonitorCreate
CALL [Device_Help]
```

## Results

```
'C' Clear if OK
    AX = monitor chain handle if Handle was "0".
    AX = 0 if handle was not "0".
'C' Set if error
    AX = error code
         Invalid monitor handle
         Not enough memory
         Monitor chain not empty
         Invalid parameter
```

## Remarks

This function may only be called at task time in the OS/2 mode only. If this function is called in the DOS mode, an invalid parameter error is returned to the device driver.

The monitor chain buffer (final_buffer) is a buffer owned by the device driver. On calling MonitorCreate, the first word of this buffer is the length of the buffer in bytes (including the first word).

When the monitor chain handle specified is 0, a new monitor chain is created. When the monitor chain handle specified is a handle that was previously returned from a call to MonitorCreate (that is, Handle < > 0), the monitor chain referenced by that handle is destroyed.

A monitor chain is a list of monitors, with a device driver monitor chain buffer address and code address as the last element on this list. Data is placed into a monitor chain through the MonWrite function; the monitor dispatcher feeds the data through all registered monitors, putting the resulting data, if any, into the specified device driver monitor chain buffer. When data is placed in this buffer the device driver's notification routine is called at task time. The device driver should initiate any necessary action in a timely fashion and return from the notification entry point without delay.

**Note:** If the MonWrite function is called at interrupt time and if the monitor chain is empty, the driver notification routine will be called at interrupt time. Under all other circumstances it is called at task time.

The MonitorCreate function establishes one of these monitor chains. The chains are created empty so that data written into them is placed immediately into the device driver's buffer.

This routine can also destroy a monitor chain if the handle parameter (AX) is non-zero. The non-zero value is the handle of the chain to remove. If the monitor chain to be removed is not empty (that is, all monitors registered with this chain have not been previously deregistered), an invalid parameter error is returned to the device driver.

A MonitorCreate call must be made before a monitor can be Registered with the chain. This can be done at any time including during the installation of the device driver at system initialization.

Notification Routine Considerations

## MonitorCreate
## Create a Monitor Chain

- The device driver's notification routine (notify_rtn) is called by the monitor dispatcher when a data record has been placed in the device driver's monitor chain buffer. The monitor dispatcher sets ES:SI = address of the device driver's monitor chain buffer and DS = the device driver's DS before calling the notification routine.
- The device driver must process the contents of the monitor chain buffer before returning to the monitor dispatcher. This entry point will be called in the OS/2 mode only.
- When the notify_rtn is called, the first word of the buffer is filled in with the length of the record just sent to the device driver. There is one notification routine call for each record.

See Chapter 8, "Character Device Monitors" for more information on the use of this call.

MonWrite passes data records to the monitors for filtering.

## Processing

```
LDS  SI,data_record_offset ; Offset of data record in DS
MOV  CX,count              ;Byte count of data record
MOV  AX,monitor_handle     ;handle for chain returned from
                                        ; previous MonitorCreate call
MOV  DH,wait_flag          ;wait/nowait flag
MOV  DL,DevHlp_MonWrite
CALL [Device_Help]
```

## Results

```
'C' Clear if OK
    AX = 0
'C' Set if error
    AX = error code
         Invalid monitor handle
         Not enough memory
         Call interrupted
```

## Remarks

This function may be called at task time or interrupt time, in the OS/2 mode or DOS mode. Wait_flag is set to 0 if the MonWrite request occurs at task or user time and the device driver wishes to have the monitor dispatcher perform the synchronization. That is, if wait_flag is set to 0, the device driver wishes to wait until the data can be placed into the monitor chain before the monitor dispatcher returns to the device driver. If wait_flag is set to 1, the device driver does not wait: if the data cannot be placed into the monitor chain, the monitor dispatcher will return immediately with the appropriate error. Wait_flag must be set to 1 if the MonWrite request occurs at interrupt time.

The DS register must be set to the device driver data segment.

The NOT ENOUGH MEMORY error will be returned to the device driver when the MonWrite call is made and the monitors are not able to receive the data. If this condition occurs at interrupt time, an overrun occurred. If it occurs at task (or user) time, the process can block.

A NOT ENOUGH MEMORY error also will be returned to the device driver when a flush record, sent to the monitors by a previous MonFlush call, was not returned to the device driver.

If the thread on which the device driver calls MonWrite blocks (the device driver specified the wait option) and is awakened because the process that owns the thread is terminating, a call interrupted error is returned to the device driver. The device driver must return the error to the caller so that the termination of the process can complete.

Each call to MonWrite will send a single complete record. The data sent by this call is considered to be a whole record. A data record must not be larger than the length of the device driver's monitor chain buffer minus two bytes.

The state of the interrupt flag is not preserved across calls to this DevHlp.

See Chapter 8, "Character Device Monitors" for more information on the use of this call.

# PhysToGDTSelector
# Map Physical Address to a GDT Selector

This function converts a 32-bit address to a GDT selector-offset pair.

## Processing

```
MOV  AX,address_high         ;32-bit physical address
MOV  BX,address_low          ;
MOV  CX,length               ;Length of segment
MOV  SI,selector             ;Selector to be setup
MOV  DL,DevHlp_PhysToGDTSelector
CALL [Device_Help]
```

## Results

```
'C' Cleared if successful
'C' Set if error
    AX = error code
         Invalid address
         Invalid selector
```

## Remarks

PhysToGDTSelector is used to provide addressability through a GDT selector to data. A bimodal device driver supports both real mode and protect mode I/O and must be able to transfer data without being dependent on the current mode of operations. In addition, the interrupt handler of a bimodal device driver must be able to address data buffers regardless of the context of the current process (the current LDT will not necessarily address the data space that contains the data buffer that the interrupt handler needs to access). The GDT selector's addressability will remain valid and the same until another PhysToGDTSelector call is made for the same selector.

The AllocGDTSelector function is used at INIT time to allocate the GDT selectors that the device driver may use with the PhysToGDTSelector. The RealToProt function (see "RealToProt Change Mode from Real to Protect Mode" on page 5-44) is used to change processor mode from real mode to protect mode, and the ProtToReal function (see "ProtToReal Change Mode from Protect to Real Mode" on page 5-35) is used to return the processor to real mode. The device driver must always return the processor to the same mode it was in when entered if either the ProtToReal or RealToProt functions are used.

PhysToGDTSelector creates selector:offset addressability for a 32-bit physical address. The selector created, however, does not represent a normal memory segment such as those usually managed by OS/2 and is more of a "fabricated segment" for private use by the device driver. Such a segment can not be passed on system calls and may only be used by the device driver to fetch data.

In the OS/2 mode PhysToUVirt converts a 32-bit physical address to a valid selector-offset pair addressable out of the current LDT. Additional information about the selector may be retained by the memory manager if special processing, based on the tag type, is required. In the DOS mode, PhysToUVirt converts a 32-bit physical address to a valid segment-offset pair, if the address is below 1 megabyte.

## Processing

```
MOV  AX,address_high  ;32-bit physical address
                      ;(or selector, if request type 2)
MOV  BX,address_low   ;
MOV  CX,length        ;Length of area (less than or
                      ; equal to 65535, 0 = 65536)
MOV  DH,request_type  ;Type of Request
                      ; 0 = get virtual address, make
                      ;     segment readable/executable,
                      ;     with Application Program Privilege
                      ; 1 = get virtual address, make
                      ;     segment readable/writable,
                      ;     with Application Program Privilege
                      ; 2 = free virtual address
                      ;     (OS/2 mode only)
                      ; 3 = get virtual address, make
                      ;     segment readable/executable,
                      ;     with I/O Privilege
                      ; 4 = get virtual address, make
                      ;     segment readable/writable,
                      ;     with I/O Privilege
                      ; 5 = get virtual address, make
                      ;     segment readable/writable,
                      ;     with Application Program Privilege
                      ;     and assign it with a tag
MOV   SI,tag type     ; Type of tag (used only on request type 5.
                      ;     preserved on all others)
                      ; 0 = foreground only video selector
MOV  DL,DevHlp_PhysToUVirt
CALL [Device_Help]
```

## Results

```
'C' Set if error
    Invalid address
'C' Cleared if successful
    ES:BX -- segment/selector-offset pair
     (for request types 0, 1, 3, and 4)
```

## Remarks

This function is typically used to provide a caller of a device driver with addressability to a fixed memory area, like ROM code and data. The device driver must know the physical address of the memory area to be addressed.

PhysToUVirt leave's its result in ES:BX. PhysToUVirt also can be used in the OS/2 mode to free a selector returned on a prior PhysToUVirt call. In the OS/2 mode, the offset returned in BX is 0 for request types 0, 1, 3, and 4. For request type 2, AX contains a selector on entry to PhysToUVirt. BX and CX are ignored.

PhysToUVirt creates selector:offset LDT addressability for a 32-bit physical address. This function is provided so a device driver can give an application process addressability to a fixed memory area, such as in the BIOS-reserved range from 640KB to 1MB. It can also be used to give a client application addressability to a device driver's data segment.

The selector created, however, does not represent a normal memory segment such as those usually managed by OS/2 and is more of a fabricated segment for private use between a device driver and

an application. Data within such a segment cannot be passed on system calls and may only be used by the receiving application to fetch data variables.

In previous releases of OS/2, all LDT selectors returned by the PhysToUVirt Device Helper routine were marked as Application Program Privilege (privilege level 3) selectors. In OS/2 Version 1.2, the device driver can specify whether the selector should be marked with Application Program Privilege or I/O Privilege (privilege level 2). This allows an LDT selector used by a dynamic link library routine, which is running with I/O privilege, to be protected from accidental modification by the application program, improving system integrity.

In the OS/2 mode, PhysToVirt converts a 32-bit address to a valid selector-offset pair. In the DOS mode, PhysToVirt converts a 32-bit address to a segment-offset pair.

## Processing

```
MOV  AX,address_high        ;32-bit physical address
MOV  BX,address_low         ;
MOV  CX,length              ;Length of segment
MOV  DH,result              ;Leave result
                            ; 0 in DS:SI
                            ; 1 in ES:DI
MOV  DL,DevHlp_PhysToVirt
CALL [Device_Help]
```

## Results

```
'C' Cleared if successful
'C' Set if error
    AX = Error code
DH  Set to 0 on input
    DS:SI Valid virtual address
    ES    No mode switch, ES is preserved.
          Mode switch, if ES contains the address of the
           device driver data segment on input, it will
           be converted to a valid virtual address.
           Otherwise, it is set to zero.
DH   Set to 1 on input
     ES:DI  Valid virtual address
     DS   No mode switch, DS is preserved.
          Mode switch, if DS contains the address of the
           device driver data segment on input, it will
           be converted to a valid virtual address.
           Otherwise, it is set to zero.
'Z'  Cleared if no change in addressing mode
'Z'  Set if addressing mode has changed
      previously stored addresses
      must be recalculated.
```

## Remarks

PhysToVirt will leave its result in either ES:DI or DS:SI, giving the device driver the ability to move strings in either direction. The returned virtual address will not remain valid if the device driver blocks or yields control. The returned virtual address also may be destroyed if the device driver routine which issues the PhysToVirt calls another routine.

On the Personal Computer AT and Personal Computer XT Model 286, if the current mode is the DOS mode and the data lies above 1 MB, then PhysToVirt will set the target segment register (DS or ES) with a special value and return to the device driver with interrupts disabled. An UnPhysToVirt DevHlp call must be made prior to exiting the device driver in this case.

The device driver must not enable interrupts or change the segment register (ES or DS - whichever contains the returned value) before the device driver has finished accessing the data area. Any change to the contents of the segment register in question will invalidate the mapping. For example, saving and restoring the value in the segment register will cause the register to refer to memory in the first megabyte of system memory. Once the device driver has finished accessing the data area, it must restore the previous interrupt state.

While pointers generated by this routine are in use, the device driver may only call another PhysToVirt request. No other DevHlp routines can be called, because they may not preserve the special DS/ES values created by the PhysToVirt.

The performance characteristics of PhysToVirt are highly variable. In the DOS mode where the entire data area lies below 1 MB or in the OS/2 mode, PhysToVirt is very fast. In the DOS mode where part or all of the data area lies above 1 MB, PhysToVirt will be very slow.

# PhysToVirt
# Map Physical Address to Virtual Address

On the PS/2, if the current mode is the DOS mode and the data lies above 1 MB, then PhysToVirt will switch into the OS/2 mode and return a selector-offset pair. This mode switch requires the use of the companion function, UnPhysToVirt, to switch back to the DOS mode.

UnPhysToVirt is required to be used any time PhysToVirt is used, with certain qualifications:

1. When use of the converted addresses is ended (no more PhysToVirts), and
2. *Before* the procedure which issued the PhysToVirt returns to its caller.

In addition, multiple PhysToVirt calls may be performed prior to issuing the UnPhysToVirt. Only one call to UnPhysToVirt is needed. When calling PhysToVirt the first time, DS must point to the device driver's device header.

PhysToVirt preserves the registers CS, SS, SP, and DS if called with DH = 1, or ES if called with DH = 0. The only exception to this case is when a mode switch takes place. If a switch to the OS/2 mode occurs because the specified address lies above 1 MB and the current mode was the DOS mode, then:

- The segment addresses in CS and SS will be set for the current mode.
- SP will be preserved.
- DS will be preserved unless it is being used for the converted address.
- ES will be set for the current mode only if it contains the data segment value of the device driver and is not being used for the converted address.

If a previous PhysToVirt had been done (using either DS or ES) with no address mode change flagged, then the PhysToVirt that requires a mode switch to the OS/2 mode causes the previously converted PhysToVirt address to be invalid for the current mode. PhysToVirt must be re-issued to recalculate the address.

When PhysToVirt is being used to recalculate an address after a mode switch is flagged, the second PhysToVirt will not cause a mode switch. The previous address will therefore be valid and preserved (so long as the recalculation uses the opposite segment register as the one that originally caused the mode switch).

Note also that in the event of a mode switch, any previously stored address pointers that contain the DS for the device driver's data segment must be stored again by the device driver. The zero flag (ZF) is set if a change in address mode occurred. In this case the device driver must recalculate and store again any buffer addresses that were previously saved.

The pool of temporary selectors used by PhysToVirt in the OS/2 mode is not dynamically extendable. The converted addresses are valid as long as the device driver does not relinquish control (Block, Yield, or RET). An interrupt handler may use converted addresses prior to its EOI, with interrupts enabled. Interrupt handlers should issue an UnPhysToVirt if necessary before making the EOI statement. If an interrupt handler needs to use converted addresses after its EOI, it must protect the converted addresses by running with interrupts disabled.

Note that the task-time strategy routine of a device driver may run enabled on a PS/2.

The segment length parameter must be set to the length of the transfer.

**Note:** For performance reasons, a device driver should try to optimize its usage of PhysToVirt and UnPhysToVirt. For the first PhysToVirt call that the device driver makes, it should pick the address that is likely to cause a mode switch and use the ES register. This would permit the mode switch to take place and retain the driver's data segment (DS).

For examples on how to use PhysToVirt and UnPhysToVirt, see the section "Using DevHlp Services for Address Conversion" on page 3-11.

This DevHlp allows a device driver to change processor mode from protect mode to real mode at interrupt time.

## Processing

```
MOV  DL,DevHlp_ProtToReal
CALL [Device_Help]
```

## Results

```
'C' Cleared if successful.  Mode has changed to real mode.
'C' Set if error.  Mode has not been changed.
    AX = error code
         Stack in High Memory
```

## Remarks

On entry, DS should be set to the device driver's data segment. On exit, the contents of ES have been changed.

RealToProt and ProtToReal require the same kernel mode restrictions that apply to PhysToVirt. All yields and blocks must be done in the same mode that the device driver was entered in. Before the yield or block, a ProtToReal must be done if a RealToProt had been used to switch to protect mode. After the yield or block, another RealToProt can be done to return to protect mode. A ProtToReal must always be done to restore mode before leaving the device driver.

Kernel mode RealToProt and ProtToReal can only be used to switch to and from protect mode when entered in real mode. RealToProt and ProtToReal can not be used to switch to real mode when the device driver is entered in protect mode.

This function is used at interrupt-time to change the processor mode from protect mode to real mode in order to service the device interrupt. The RealToProt DevHlp function is used to switch the processor mode back from real mode to protect mode. See "RealToProt Change Mode from Real to Protect Mode" on page 5-44.

Once the processing is complete, the device driver must return the processor to the original mode the processor was in when the device driver was entered. For example:

```
        .
        .
        .

Device driver entered in Protect Mode.

Call DevHlp_ProtToReal - change to real mode.
        .
        .
        .

Device processing in Real Mode
        .
        .
        .

Call DevHlp_RealToProt - change back to protect mode.

Device driver returns or exits.
```

# ProtToReal
# Change Mode from Protect to Real Mode

The following shows an example of how to determine whether the processor must be switched from protect mode to real mode and back by checking the MSW:

```
        smsw   ax          ; get current msw
        push   ax          ; save original msw
        rcr    ax,1        ; pe bit -> cf
        jnc    rm1         ; already real mode?
        DevHlp ProtToReal  ; no, switch to real mode
rm1:    .
        .                  ; process in real mode
        .
        pop    ax          ; get original msw
        rcr    ax,1        ; pe bit -> cf
        jnc    rm2         ; originally in real mode?
        DevHlp RealToProt  ; no, switch back to protect mode
rm2:    .
        .
```

Device drivers are restricted from issuing this call with a stack segment in high memory (for example, when the device driver is in kernel mode and the device is opened from a protect mode screen group).

PullParticular pulls the specified packet from the selected request packet linked list. If the packet is not found, then an indicator is set on return.

## Processing

```
MOV  SI,OFFSET DS:queue        ;Location of queue head (should
                               ; match PushRequest value)
LES  BX,request_packet         ;Pointer to request packet.
MOV  DL,DevHlp_PullParticular
CALL [Device_Help]
```

## Results

'C' Cleared if no error.
'C' Set if the specified request is not found.

## Remarks

A device driver uses the PushReqPacket and PullReqPacket DevHlps to maintain a work queue for each of its devices. PullParticular is used to remove a specific request packet from the work queue, typically for the case where a process has terminated before finishing its I/O.

PullParticular may also be used to remove request packets that were allocated by an AllocReqPacket from the request packet linked list.

# PullReqPacket
# Remove Request From List

PullReqPacket pulls the next waiting request packet from the selected request packet linked list. If there is no packet in the list, then an indicator is set on return.

## Processing

```
MOV  SI,OFFSET DS:queue       ;Location of queue head (should
                              ; match PushReqPacket value)

MOV  DL,DevHlp_PullReqPacket
CALL [Device_Help]
```

## Results

```
'C' Set if there is no request.
'C' Cleared if no error.
    ES:BX  Pointer to request packet.
```

## Remarks

A device driver uses the PushReqPacket and PullReqPacket DevHlps to maintain a work queue for each of its devices/units. The device driver must provide the storage for the DWORD work queue head, which defines the start of the request packet linked list. The work queue head must be initialized to 0.

PullReqPacket may also be used to remove request packets that were allocated by an AllocReqPacket from the request packet queue.

PushReqPacket adds the current device request packet to the linked list of packets to be executed by the device driver.

## Processing

```
MOV  SI,OFFSET DS:queue      ;Location of the DWORD queue head
                             ; (which points to the first
                             ; request in the list.)
LES  BX,request_packet       ;Pointer to request
                             ; packet.

MOV  DL,DevHlp_PushReqPacket
CALL [Device_Help]
```

## Results

None

## Remarks

A device driver uses the PushReqPacket and PullReqPacket DevHlps to maintain a work queue for each of its devices. The device driver must provide the storage for the DWORD work queue head, which defines the start of the request packet linked list. The work queue head must be initialized to 0.

The device driver task-time thread should add all incoming read/write requests to its request list. The driver task-time thread should then determine whether the interrupt-time thread is active, and if not, it should send the request to the device. Because the device may be active at this point, the driver task-time thread must turn off interrupts before calling the device; otherwise a window exists in which the device finishes before the packet is put on the list.

PushReqPacket may also be used to place request packets that were allocated by an AllocReqPacket in the request packet work queue.

# QueueFlush
# Clear Character Queue

QueueFlush clears the character queue structure that is specified (it empties the buffer).

## Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure
                             ; to be flushed.  (The Qsize
                             ; field must be set up.)
MOV  DL,DevHlp_QueueFlush
CALL [Device_Help]
```

## Results

None

## Remarks

QueueFlush operates on the simple character queue structure initialized by QueueInit.

QueueInit initializes the specified character queue structure.

## Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure
                             ; to be initialized.  (The
                             ; Qsize field must be set up.)
MOV  DL,DevHlp_QueueInit
CALL [Device_Help]
```

## Results

None

## Remarks

QueueInit must be called before any other queue manipulation subroutine.

Prior to this call, the device driver must allocate the character queue buffer with the following queue header and initialize the Qsize field.

For example:

```
Qsize     DW    ?    ;size of queue in bytes
Qchrout   DW    ?    ;index of next char out
Qcount    DW    ?    ;count of chars in the queue
Qbase     DB    ?    ;start of queue buffer
```

# QueueRead
# Read a Character From a Queue

QueueRead returns and removes a character from the beginning of the specified character queue structure. If the queue is empty, an indicator is set.

## Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure.
MOV  DL,DevHlp_QueueRead
CALL [Device_Help]
```

## Results

```
'C' Set if the queue is empty,
'C' Cleared otherwise.
AL  The character read from the queue.
```

## Remarks

QueueRead operates on the simple character queue structure initialized by QueueInit.

QueueWrite adds a character at the end of the specified character queue structure. If the queue is full, an indicator is set.

## Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure.
MOV  AL,char                 ;Character to insert at the end
                             ; of the queue.
MOV  DL,DevHlp_QueueWrite
CALL [Device_Help]
```

## Results

```
'C' Clear if character stored successfully.
'C' Set if queue is full.
```

## Remarks

QueueWrite operates on the simple character queue structure initialized by QueueInit.

# RealToProt
# Change Mode from Real to Protect Mode

This DevHlp allows a device driver to change processor mode from real mode to protect mode at interrupt time.

## Processing

```
MOV  DL,DevHlp_RealToProt
CALL [Device_Help]
```

## Results

```
'C' Cleared if successful
    Mode has changed to protect mode.
'C' Set if error
    Mode has not been changed.
```

## Remarks

On entry, DS should be set to the device driver's data segment. On exit, the contents of ES have been changed.

RealToProt and ProtToReal require the same kernel mode restrictions that apply to PhysToVirt. All yields and blocks must be done in the same mode that the device driver was entered in. Before the yield or block, a ProtToReal must be done if a RealToProt had been used to switch to protect mode. After the yield or block, another RealToProt can be done to return to protect mode. A ProtToReal must always be done to restore mode before leaving the device driver.

Kernel mode RealToProt and ProtToReal can only be used to switch to and from protect mode when entered in real mode. RealToProt and ProtToReal can not be used to switch to real mode when the device driver is entered in protect mode.

This function is used at interrupt-time to change the processor mode from real mode to protect mode in order to process the device interrupt. The DevHlp function ProtToReal is used to switch the processor mode back from protect mode to real mode. See "ProtToReal Change Mode from Protect to Real Mode" on page 5-35.

Once the processing is complete, the device driver must return the processor to the original mode that the processor was in when the device driver was entered. For example:

```
        .
        .
        .
Device driver entered in Real Mode.

Call DevHlp_RealToProt - change to protect mode.
        .
        .
        .

Device processing in Protect Mode
        .
        .
        .

Call DevHlp_ProtToReal - change back to real mode.

Device driver returns or exits.
```

The following shows an example of how to determine whether the processor must be switched from real mode to protect mode and back by checking the MSW:

```
     smsw   ax          ; get current msw
     push   ax          ; save original msw
     rcr    ax,1        ; pe bit -> cf
     jc     pm1         ; already protect mode?
     DevHlp RealToProt  ; no, switch to protect mode
pm1: .
     .                  ; process in protect mode
     .
     pop    ax          ; get original msw
     rcr    ax,1        ; pe bit -> cf
     jc     pm2         ; originally in protect mode?
     DevHlp ProtToReal  ; no, switch back to real mode
pm2: .
     .
```

# Register
# Add Monitor

Register adds a monitor to the chain of monitors for a class of device.

## Processing

```
LES  SI,input_buffer         ;Address of input buffer
MOV  DI,output_buffer_offset ;Offset of output buffer
MOV  CX,monitor_PID          ;Process ID of monitor task
MOV  AX,monitor_handle       ;handle for chain returned from
                                        ; previous MonitorCreate call
MOV  DH,placement_flag       ;High or Low place in chain
MOV  DL,DevHlp_Register
CALL [Device_Help]
```

## Results

```
'C' Clear no error
'C' Set if error
    AX = error code
         Invalid monitor handle
         Invalid parameter
         Not enough memory
```

## Remarks

This function may only be called at task time in the OS/2 mode only. If this function is called in the DOS mode, an invalid parameter error is returned to the device driver.

A monitor chain must have previously been created with MonitorCreate.

A single process may register more than one monitor (with "different" input and output buffers) with the same monitor chain.

The first word of each of the input and output buffers must contain the length in bytes (length word inclusive) of the buffers. The length of the monitor's input and output buffers must be greater than the length of the device driver's monitor chain buffer plus 20 bytes.

The input buffer, output buffer offset, and placement flag are supplied to the device driver by the monitor application that is requesting monitor registration (that is, calling DosMonReg).

The device driver must identify the monitor chain with the monitor handle returned from a previous MonitorCreate call. The device driver can determine the PID of the requesting monitor task from the Local InfoSeg. See "GetDOSVar This routine is used to return the address of a kernel variable." on page 5-21.

See Chapter 8, "Character Device Monitors" for more information on the use of this call.

Indicates the expected stack usage of device driver to interrupt manager

## Processing

```
MOV BX,OFFSET DS:StackUsage
MOV  DL,DevHlp_RegisterStackUsage
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
'C' Set if stack usage exceeds system maximum.
    The device driver must deinstall itself.
```

## Remarks

The StackUsage data structure has the following format:

```
StackUsage     STRUC
 SU_cbStruct   DW 14     ; number of bytes in structure including itself.
 SU_flags      DW ?      ; bit 0x0001 'on' indicates that the interrupt
                         ; procedure enables interrupts.  All other bits
                         ; are reserved.
 SU_iIRQ       DW ?      ; IRQ of interrupt handler that is being
                         ; described by the following data.
 SU_cbStackCLI DW ?      ; number of bytes of stack used in the
                         ; interrupt procedure when interrupts are disabled.
 SU_cbStackSTI DW ?      ; number of bytes of stack used after interrupt
                         ; procedure enables interrupts.
 SU_cbStackEOI DW ?      ; number of bytes of stack used after interrupt
                         ; procedure issues EOI.
 SU_cNest      DW ?      ; maximum number of levels that the device
                         ; driver expects to nest.
StackUsage     ENDS
```

If the device driver interrupt routines nest greater than the specified number, the interrupt manager will disable the IRQ at the PIC for the remainder of the boot session.

A device must issue this DevHlp once for each IRQ that it services.

OS/2 supports a total of 8KB of interrupt stack.

# ResetTimer
# Reset Timer Handler

ResetTimer removes a timer handler for the device driver.

## Processing

```
MOV  AX,Offset cs:timer_handler        ;offset to timer handler
MOV  DL,DevHlp_ResetTimer
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error
'C' Set if error
    Invalid handler
```

## Remarks

This function removes a timer handler from the list of timer handlers. Timer handlers are analogous to the user timer interrupt (Int 1CH). Refer to "TickCount Modify timer" on page 5-63 for a discussion on timer handlers.

DS should be set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it should restore DS to the original value.

ROMCritSection flags a critical section of execution in a DOS mode software interrupt handler to prevent the DOS mode from being suspended in the background.

## Processing

```
MOV  AL,enter_or_exit        ;Critical Section flag
                             ; = 0  exit
                             ; < > 0  enter
MOV  DL,DevHlp_ROMCritSection
CALL [Device_Help]
```

## Results

None.

## Remarks

This service is called by a device driver's DOS mode software interrupt handler.

Sections of ROM BIOS code must be protected from preemption. Preemption occurs when a user switches away from the DOS mode. This causes the DOS mode to be suspended in background. However, some I/O processing cannot tolerate being suspended. Specific examples are the printer (BIOS INT 17H), disk (BIOS INT 13H), and screen (BIOS INT 10H). It is the responsibility of the OS/2 device driver to intercept the appropriate ROM BIOS interrupt and issue the DevHlp function call, ROMCritSection, to protect the ROM BIOS critical section of execution.

**Note:** When the OS/2 device driver issues ROMCritSection to "enter" a ROM BIOS critical section, the user will not be able to switch away from the screen of the DOS mode to another screen. This has potential problems for the user. For example, if some DOS mode terminate-and-stay-resident program takes control while the CPU is executing the ROM BIOS, the time spent in the ROM BIOS critical section will be longer, and the user will be unable to switch tasks. The worst case is that the terminate-and-stay-resident application is interactive, never allowing the OS/2 device driver to issue the "exit" from critical section and never allowing the user to switch away from the screen of the DOS mode until the user terminates the application.

# Run
# Release Blocked Thread

This is the companion routine to Block. When Run is called, it awakens *all* threads that were blocked for this particular event identifier.

## Processing

```
MOV  BX,event_id_low        ;Low word of event identifier.
MOV  AX,event_id_high       ;High word of event identifier.
MOV  DL,DevHlp_Run
CALL [Device_Help]
```

## Results

```
AX = count of those awakened.
'Z'  set according to AX.
```

## Remarks

Run returns immediately to its caller; the awakened threads will be run at the next available opportunity. Run is often called at interrupt time.

Refer to "Block This Thread From Running" on page 5-13 for a detailed discussion.

This service is provided to the clock device driver to allow it to obtain a pointer to the address of the system's clock tick handler, SchedClock. SchedClock must be called on each occurrence of a periodic clock tick.

## Processing

```
MOV  AX,Pointer_Save          ;Offset in DS to a DWORD
                              ; where the pointer will
                              ; be returned
MOV  DL,DevHlp_SchedClockAddr
CALL [Device_Help]
```

## Results

```
Pointer_Save will contain the address of
the system tick handler.
```

## Remarks

The clock device driver calls this DevHlp service during the clock device driver's initialization. For input to this DevHlp, the clock device driver must ensure that its DS points to the device driver's data segment and supply the offset to a DWORD area. The DevHlp service will then fill in the device driver's save area with a bimodal pointer to a DWORD in system memory. The DWORD in system memory contains the pointer to the SchedClock routine. This is illustrated in the following figure:

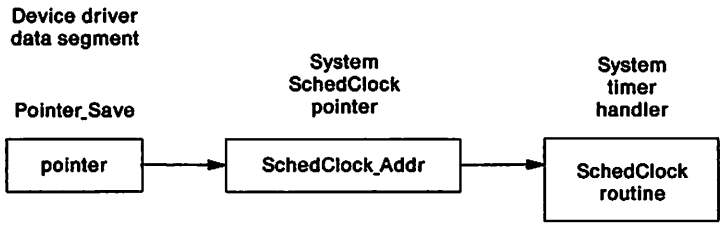



Figure 5-1. DWORD in system memory

The pointer that is returned to the device driver in Pointer_Save is a bimodal pointer which is valid in either the DOS mode or the OS/2 mode. The device driver does not need to perform any conversions on this pointer for DOS mode or OS/2 mode operations. The device driver can then use the pointer it has in Pointer_Save to call SchedClock. SchedClock is called by the clock device driver with a CALL FAR INDIRECT, using the pointer to the area which contains the actual address of the SchedClock routine.

# SchedClockAddr
# Get system clock routine

SchedClock must be called at interrupt time for each periodic clock tick to indicate the passage of system time. The "tick" is then dispersed to the appropriate components of the system. A definition of the interface to SchedClock follows.

```
MOV  AL,millisecs            ;Milliseconds since last call
MOV  DH,EOIflag              ;Indicator of EOI
                             ; = 0 prior to EOI
                             ; = 1 after EOI
CALL [pointer]               ;Pointer to the area which
                             ; contains the actual address
                             ; of SchedClock
```

The clock device driver's interrupt handler must run with interrupts enabled as the convention, prior to issuing the EOI for the timer interrupt. Any critical processing, such as updating the fraction-of-seconds count, must be done prior to calling SchedClock. SchedClock must then be called to allow system processing prior to the dismissal of the interrupt. When SchedClock returns, the clock device driver must issue the EOI and call SchedClock again. Note that once the EOI has been issued, the device driver's interrupt handler may be reentered. The DevHlp SchedClock is also reentrant.

The device driver must not get the actual address of the SchedClock routine but instead use the pointer returned by the DevHlp call.

This function releases a semaphore and restarts any blocked threads waiting on the semaphore.

## Processing

```
MOV  BX,sem_handle_low    ;Semaphore handle
MOV  AX,sem_handle_high   ;
MOV  DL,DevHlp_SemClear
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error
'C' Set if error
    AX = error code
         error_invalid _handle
         error_excl_sem_already_owned
         error_invalid_at_interrupt_time
         error_protection_violation
```

## Remarks

A device driver may clear either a RAM semaphore or a system semaphore. The device driver may obtain (own) a semaphore through SemRequest.

The semaphore handle for a RAM semaphore is the virtual address of the doubleword of storage allocated for the semaphore. Virtual address is used as a generic term for addresses: segment:offset for the DOS mode; selector:offset for the OS/2 mode. To access a RAM semaphore at interrupt time, the device driver must locate the semaphore in the device driver's data segment (DS).

For a system semaphore, the handle must be passed to the device driver by the caller by way of a generic IOCtl. The device driver must convert the caller's handle to a system handle with SemHandle.

A RAM semaphore can be cleared at interrupt time only if it is in storage that is directly addressable by the device driver, that is, in the device driver's data segment.

# SemHandle
# Obtain a Semaphore Handle

This function provides a semaphore handle to the device driver.

## Processing

```
MOV  BX,sem_key_low    ;Semaphore identifier
MOV  AX,sem_key_high   ;
MOV  DH,usage_flag     ;Indicates if in use
                       ; = 0 not-in-use
                       ; = 1 in-use
MOV  DL,DevHlp_SemHandle
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error
    AX:BX set to the system handle.
'C' Set if error
    AX contains error code.
    Invalid handle for the semaphore if DH = 1.
```

## Remarks

This function is used to convert the semaphore handle (or user "key") provided by the caller of the device driver to a system handle that the device driver may use. This handle then becomes the "key" that the device driver uses to reference the System Semaphore. This allows the System Semaphore to be referenced at interrupt time by the device driver. This "key" is also used when the device driver is finished with the system semaphore. The device driver must call SemHandle with the usage_flag indicating that the device driver is finished with the system semaphore.

SemHandle is called at task time to indicate that the system semaphore is IN-USE, and is called at either task time or interrupt time to indicate that the system semaphore is NOT-IN-USE. IN-USE means that the device driver may be referencing the System Semaphore. NOT-IN-USE means that the device driver has finished using the system semaphore and will not be referencing it again.

The "key" of a RAM semaphore is its virtual address, where virtual address is the generic term for both DOS mode and OS/2 mode address forms (segment:offset, selector:offset). SemHandle may be used for RAM semaphores. Because RAM semaphores have no system handles, SemHandle will simply return the RAM semaphore "key" back to the caller.

A device driver can determine that a semaphore is a RAM semaphore if the key remains unchanged upon return from the SemHandle function. If the key returned from SemHandle is different than the one passed to the function, then the device driver can determine that it is a handle for a System semaphore.

If carry is returned from this function, the device driver should issue the DevHlp VerifyAccess request with TypeOfAccess of Read/Write indicated before assuming this is a RAM semaphore. If a Ram semaphore is to be used, it must be accessed only at task time unless it is in locked storage.

It is necessary to call SemHandle at task time to indicate that a System Semaphore is IN-USE because:

1. The caller-supplied semaphore handle refers to task-specific system semaphore structures. These structures are not available at interrupt time, so SemHandle converts the task-specific handle to a system-specific handle. For uniformity, the other semaphore DevHlp functions accept only system-specific handles regardless of the mode (task time or interrupt time).
2. An application could delete a system semaphore while the device driver is using it. If a second application were to create a system semaphore soon after, the system structure used by the original semaphore could be reassigned. A device driver which tried to manipulate the original process's semaphore would inadvertently manipulate the new process's semaphore. Therefore, the SemHandle IN-USE indicator increases a counter so that, although the calling thread may still delete its task-specific reference to the semaphore, the semaphore remains in the system structures.

A device driver must subsequently call SemHandle with NOT-IN-USE when the semaphore use is done so that the system semaphore structure can be freed. There must be a call to indicate NOT-IN-USE to match every call to indicate IN-USE (one-to-one relationship).

The state of the interrupt flag is not preserved across calls to this DevHlp.

# SemRequest
# Claim a Semaphore

This function claims a semaphore. If the semaphore is already owned, the thread in the device driver is blocked until the semaphore is released or until a timeout occurs.

## Processing

```
MOV  BX,sem_handle_low    ;Semaphore handle
MOV  AX,sem_handle_high   ;
MOV  CX,sem_timeout_low   ;Timeout value
MOV  DI,sem_timeout_high  ;in milliseconds
                          ; = -1 wait forever
                          ; =  0 no wait if sem owned
                          ; >  0 timeout
MOV  DL,DevHlp_SemRequest
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error.
'C' Set if error.
    AX = error code.
         error_sem_timeout
         error_sem_owner_died
         error_invalid_handle
         error_too_many_sem_requests
         error_interrupted
         error_protection_violation
```

## Remarks

SemRequest checks the state of the semaphore. If it is unowned, SemRequest marks it "owned" and returns immediately to the caller. If the semaphore is owned, SemRequest will optionally block the device driver thread until the semaphore is unowned, then try again. The timeout parameter is used to place an upper limit on the amount of time to block before returning to the requesting device driver thread.

SemClear is used at either task time or interrupt time to release the semaphore.

The semaphore handle for a RAM semaphore is the virtual address of the doubleword of storage allocated for the semaphore. Virtual address is used as a generic term for addresses: segment:offset for the DOS mode; selector:offset for the OS/2 mode. To access a RAM semaphore at interrupt time, the device driver must locate the semaphore in the device driver's data segment (DS).

For a system semaphore, the handle must be passed to the device driver by the caller through a generic IOCtl. The device driver must convert the caller's handle to a system handle with SemHandle.

Note that this service is valid in user mode only for RAM semaphores. System semaphores are not available in user mode by device drivers.

The state of the interrupt flag is not preserved across calls to this DevHlp.

This routine is called by a device driver to indicate the occurrence of an event.

## Processing

```
MOV  AH,event                ;Event being signalled
MOV  BX,argument             ;Parameter for the event
                             ; being signalled
MOV  DL,DevHlp_SendEvent
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error.
'C' Set if error sending signal.
```

## Remarks

The events are defined as:

- Session manager hot key from the mouse
  - Event = 0 (Reserved)
  - argument = 2-byte time stamp
    Where the high-order byte is seconds and the low-order byte is hundreths of seconds.
- Ctrl + Break
  - event = 1
  - argument = 0 (Reserved)
- Ctrl + C
  - event = 2
  - argument = 0 (Reserved)
- Ctrl + NumLock
  - event = 3
  - argument = Foreground session number
- Ctrl + PrtSc
  - event = 4
  - argument = 0 (Reserved)
- Shift + PrtSc
  - event = 5
  - argument = 0 (Reserved)
- Session Manager Hot Key from the Keyboard
  - event = 6
  - argument = Hot Key ID

  The keyboard device driver uses the Hot Key ID which was set by way of keyboard IOCtl 56H (SET SESSION MANAGER HOT KEY).
- Reboot Key Sequence from the Keyboard
  - event = 7
  - argument = 0 (Reserved)

# SetIRQ
# Set Hardware Interrupt Handler

This service is used to set a hardware interrupt vector to the device driver interrupt handler.

## Processing

```
MOV  AX,Offset CS:handler ;Interrupt handler offset
MOV  BX,IRQnum            ;Interrupt level number (0 - FH)
MOV DH,Shared_Int         ;Interrupt sharing (=0 not shared, =1 shared)
MOV  DL,DevHlp_SetIRQ
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
'C' Set if error
      IRQ is not available.
```

## Remarks

The attempt to register an interrupt handler for an IRQ to be Shared will fail if the IRQ is already owned by another device driver as Not Shared, owned by a DOS mode interrupt handler, or is the IRQ used to cascade the slave 8259 interrupt controller (IRQ 2).

The attempt to register an interrupt handler for an IRQ to be Not Shared will fail if the the IRQ is already owned by another device driver as Shared or Not Shared, owned by a DOS mode interrupt handler, or is the IRQ used to cascade the slave 8259 interrupt controller.

DS should be set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it should restore DS to its original value.

The IRQnum value is range checked and 'C' is set if not from 0 to 0FH.

Hardware interrupt sharing is not supported on all systems. A SetIRQ request to share an interrupt level on a system where sharing is not supported will return an error. See the section on Hardware Interrupt Management in this book for a discussion of the limitations on hardware interrupt sharing and the systems supported.

This service is used to replace a DOS mode software interrupt handler with a handler from the device driver. This service returns a DOS mode pointer to the previous software interrupt handler for chaining.

## Processing

```
MOV  AX,OFFSET CS:handler    ;Interrupt handler offset
MOV  BX,intnum               ;Interrupt number
MOV  SI,OFFSET CS:saveDS     ;offset in CS of WORD
                             ; to save DOS mode DS
MOV  DL,DevHlp_SetROMVector
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
    AX:DX = DOS mode pointer to previous header.
    CS:[SI] = DOS mode DS of device driver.
'C' Set if error
    Invalid interrupt number.
```

## Remarks

The device driver can register a software interrupt handler for a DOS mode software interrupt. This is typically done to intercept a ROM BIOS software interrupt, which allows the device driver to perform any processing needed to coordinate device I/O between the driver and ROM BIOS.

The device driver may not register a software interrupt handler for any of the interrupt vectors reserved for hardware interrupts. The reserved interrupt numbers are:

08 - 0FH
50 - 57H
70 - 77H

The device driver's software interrupt handler for the DOS mode software interrupt receives control directly when the interrupt occurs. Consequently, the DS register is not set up for the handler on entry. The handler must set the DS register with the value that SetROMVector had previously saved in the CS:[SI] location.

When calling SetROMVector, the device driver must ensure that DS is set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it must restore DS to the original value.

# SetTimer
# Set Timer Handler

SetTimer adds a timer handler to the list of timer handlers to be called on a timer tick.

## Processing

```
MOV  AX,OFFSET CS:timer_handler        ;Offset of timer handler.
MOV  DL,DevHlp_SetTimer
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error.
'C' Set if error.
    Timer handler disallowed (maximum number of
     handlers reached or timer handler already set).
```

## Remarks

The DevHlp SetTimer is a subset of the DevHlp TickCount.

This function allows a device driver to add a timer handler to a list of timer handlers called on every timer tick. A device driver may use a timer handler to drive a non-interrupt device instead of using timeouts with the Block and Run services. Block and Run are costly on a character-by-character basis; the cost is one or more task switches for each character I/O. Timer handlers are required to save and restore registers.

A maximum of 32 different timer handlers are available in the system.

While a timer handler is in the format of a FAR CALL/RETURN routine (when it is finished processing, it performs a return), it operates in Interrupt State. The timer handler is analogous to the user timer (Int 1CH) handler. Care should be taken not to remain in the handler very long.

DS should be set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it should restore DS to the original value.

Timer handlers are responsible for saving and restoring registers on entry and exit.

This routine is used by block (disk) device drivers to add a new request to their work queue. This routine inserts the request packet in the linked list of request packets in the order of starting sector number.

## Processing

```
MOV  SI,OFFSET DS:queue      ;Location to DWORD queue
                             ; head (which points to
                             ; first request).  It
                             ; should be initialized
                             ; to 0.
LES  BX,request_packet       ;Pointer to request
                             ; packet.
MOV  DL,DevHlp_SortReqPacket
CALL [Device_Help]
```

## Results

None

## Remarks

The sorting by sector number is aimed at reducing the length and number of head seeks. This is a simple algorithm and does not account for multiple heads on the media or for target drive in the request packet. SortReqPacket inserts the current request packet into the specified linked list of packets, sorted by starting sector number.

SortReqPacket may be used to place request packets that were allocated by an AllocReqPacket in the request packet queue.

# TCYield
# Yield the CPU

This function is similar to the Yield function, except that the CPU may only be yielded to a time-critical thread, if one is available.

## Processing

```
MOV  DL,DevHlp_TCYield
CALL [Device_Help]
```

## Results

None

## Remarks

It is not necessary for the device driver to do both a Yield and a TCYield. The TCYield function is a subset of the Yield function.

The one part of the kernel that can take a lot of CPU time is in device drivers, particularly those that perform program I/O on long strings of data or that poll a device. These device drivers periodically should check the TCYield Flag and call the TCYield function to yield the CPU to a time-critical thread.

The location of the TCYield Flag is obtained from the GetDOSVar call.

For performance reasons, the device driver should check the TCYield Flag once every 3 milliseconds. If the flag is set, then the device driver should call TCYield.

Because the device driver may relinquish control of the CPU, the device driver should not assume that the state of the interrupt flag will be preserved across a call to TCYield.

TickCount will register a new timer handler or modify a previously registered timer handler to be called on every N timer ticks instead of every timer tick.

## Processing

```
MOV AX,OFFSET CS:timer_handler  ;offset to timer handler
MOV BX,count                    ;N tick counts (0-FFFF)
                                ; 0 means FFFFH+1 ticks
MOV  DL,DevHlp_TickCount
CALL [Device_Help]
```

## Results

```
'C' Cleared if no error
'C' Set if error
    Timer handler cannot be modified or set.
```

## Remarks

A device driver may use a timer handler to drive a non-interrupt device instead of using timeouts with the Block and Run services. Block and Run are costly on a character-by-character basis; the cost is one or more task switches for each character I/O. Timer handlers are required to save and restore registers.

While a timer handler is in the format of a FAR CALL/RET routine, it operates at interrupt time. The timer handler is analogous to the user timer (INT 1CH) handler. Care must be taken not to remain in the handler very long.

For a new timer handler, TickCount will register the timer handler to be called on every N timer ticks instead of every timer tick.

For a previously registered timer handler, TickCount changes the number of ticks that must take place before the timer handler gets control. This will allow device drivers to support the timeout function without needing to count ticks.

At task time, this DevHlp may be used to modify a timer handler registered through SetTimer or may be used to register a new timer handler that is initially invoked every N ticks.

In user mode (task time), this DevHlp may be used only to modify a timer handler *already* registered.

In interrupt mode (interrupt time), this DevHlp may be used only to modify a timer handler *already* registered. This allows an interrupt handler to reset the timing condition at interrupt time.

Note that SetTimer sets a default of N ticks to 1. Multiple TickCount requests may be issued for a given timer handler, but only the last TickCount setting will be in effect.

TickCount affects only the specified registered timer handler. It has no effect on other timer handlers.

DS should be set to the device driver's data segment. If the device driver did a PhysToVirt referencing the DS register, it should restore DS to the original value.

Timer handlers are responsible for saving and restoring registers on entry and exit.

A maximum of 32 different timer handlers are available in the system.

# Unlock
# Unlock Memory Segment

This service unlocks a locked memory segment in any state.

## Processing

```
MOV  BX,lock_handle_low       ;Handle for segment
MOV  AX,lock_handle_high      ; returned by Lock
MOV  DL,DevHlp_Unlock
CALL [Device_Help]
```

## Results

'C' Cleared if segment unlocked
Set if otherwise.

## Remarks

none.

UnPhysToVirt is *required* to mark completion of address conversion from PhysToVirts.

## Processing

```
MOV  DL,DevHlp_UnPhysToVirt
CALL [Device_Help]
```

## Results

```
'Z' Cleared if no address mode change.
'Z' Set if address mode change.
```

## Remarks

This function forms part of the structure of mode-dependent addressing on behalf of a device driver, relieving it of the need to understand the CPU mode and the subsequent effects on accessing memory.

On the PS/2, if the converted address caused a switch to the OS/2 mode in PhysToVirt, UnPhysToVirt will switch back to the DOS mode.

UnPhysToVirt must be called by the same procedure that issued the PhysToVirt when use of converted addresses is completed and *before* the procedure returns to its caller. The procedure that called PhysToVirt may call other procedures before calling UnPhysToVirt. Multiple PhysToVirt calls may be issued prior to issuing the UnPhysToVirt. Only one call to UnPhysToVirt is needed.

The ZF is set if a mode switch occurred. This allows the device driver to recalculate any stored pointers that were not used in the data transfer operations with the PhysToVirt.

UnPhysToVirt, if switched to the DOS mode, will reset the registers CS and SS to the DOS mode.

SP will be preserved.

DS will be reset to the device driver's data segment.

Note that the addresses that caused the switch into the OS/2 mode cannot be preserved or converted for the DOS mode.

The ES register will not be preserved. The ES register is also set to the device driver data segment.

For examples on how to use PhysToVirt and UnPhysToVirt, see the section "Using DevHlp Services for Address Conversion" on page 3-11.

# UnSetIRQ
# Remove Hardware Interrupt Handler

This routine removes the current hardware interrupt handler.

## Processing

```
MOV  BX,IRQnum              ;IRQ Interrupt number
                            ; ( 0 - F )
MOV  DL,DevHlp_UnSetIRQ
CALL [Device_Help]
```

## Results

```
'C' Set if caller is not the owner of
    or one of the owners of the IRQ.
```

## Remarks

DS must point to the device driver's data segment on entry.

This routine verifies that the user process has the correct access rights for the memory that it passed to the device driver. If the process does not have the needed access rights to the memory, then it will ve terminated. If it does have the needed access rights, these rights are guaranteed to remain valid until the device driver exits its strategy routine.

## Processing

```
MOV  AX,Segment_@           ;Selector or segment
MOV  CX,MemLength           ;Length of memory area
                            ; in bytes (0 means 64 KB)
MOV  DI,Mem_Offset          ;Offset to memory area
MOV  DH,TypeOfAccess        ;Verify access for
                            ; = 0  read access
                            ; = 1  read/write access
MOV  DL,DevHlp_VerifyAccess
CALL [Device_Help]
```

## Results

```
'C' Clear if no error
    Access verified.
'C' Set if error
    Access attempt failed.
```

## Remarks

A device driver can receive addresses to memory as part of a generic IOCtl request from a process. Because the operating system cannot verify addresses imbedded in the IOCtl command, the device driver must request verification in order to prevent itself from accidentally erasing memory on behalf of a user process. If the verification test fails, then VerifyAccess will terminate the process.

Note that verification may only take place in the OS/2 mode. If VerifyAccess is called in the DOS mode, it will return stating that the memory is accessible.

Once the device driver has verified the process has the needed access to addresses of interest, it does not need to repeat the verification until it yields the CPU. When the device driver yields the CPU, all address access verifications it has done become unreliable, except for segments which have been locked. The device driver could yield the CPU by accessing a not-present-segment, exiting its strategy routine, or calling a devhlp service which yields while performing the service.

If the device driver plans to Lock the segment into memory, it should perform the lock before the VerifyAccess. If the selector is invalid, Lock will return an error. The device driver can then call VerifyAccess and it will terminate the process if the address insufficient access permissions for the requested I/O operation.

**Note:** If the device driver SUCCESSFULLY calls the Lock DevHlp before calling VerifyAccess and the VerifyAccess call terminates the application, the device driver MUST UnLock the segment before returning.

# VirtToPhys
# Map Virtual Address to Physical Address

When in the OS/2 mode, it converts a selector-offset pair to a 32-bit physical address. When in the DOS mode, VirtToPhys converts a segment-offset pair to a 32-bit physical address.

## Processing

```
LDS  SI,address              ;Virtual address:
                             ; segment:offset or selector:offset
MOV  DL,DevHlp_VirtToPhys
CALL [Device_Help]
```

## Results

```
'C' Cleared to indicate no error.
    AX:BX  Physical address: 32-bit number.
```

## Remarks

The virtual address should be locked by way of the DevHlp service Lock prior to invoking this function, if the segment is not known to be locked already.

This function is typically used to convert a virtual address supplied by a process by way of a generic IOCtl, in order that the memory may be accessed at interrupt time.

This routine yields the CPU to a scheduled thread of equal or higher priority.

## Processing

```
MOV  DL,DevHlp_Yield
CALL [Device_Help]
```

## Results

None

## Remarks

OS/2 is designed so that the CPU never is scheduled preemptively while in kernel mode. In general, the kernel either performs its job and exits quickly, or it blocks waiting for (usually) I/O or (occasionally) a resource. It is not necessary for the device driver to do both a Yield and a TCYield; the Yield function is a superset of the TCYield function.

The one part of the kernel that can take a lot of CPU time is in device drivers, particularly those that perform program I/O on long strings of data or poll the device. These drivers should periodically check the Yield Flag and call the Yield function to yield the CPU if another process needs it. Much of the time the context won't switch; Yield switches context only if an equal or higher priority thread is scheduled to run.

The address of the Yield Flag is obtained from the GetDOSVar call. Refer to "GetDOSVar This routine is used to return the address of a kernel variable." on page 5-21.

For performance reasons, the device driver should check the Yield Flag once every 3 milliseconds. If the flag is set, then the device driver should call Yield.

Because the device driver may relinquish control of the CPU to another thread, the device driver should not assume that the state of the interrupt flag will be preserved across a call to Yield.

# Chapter 6. Device Drivers

The device drivers provided with OS/2 service requests in both the DOS mode and the OS/2 mode. Where appropriate, OS/2 device drivers provide a queued request interface rather than the serial request design of DOS device drivers. OS/2 device drivers support multitasking.

OS/2 device driver functions are not directly invoked by applications. The OS/2 File System and Device Manager provide the application interface. In addition, the IOCtl interface allows an application or subsystem to send device-specific control commands to the device driver. The IOCtl interface for OS/2 mode applications or subsystems is the DosDevIOCtl function call. The IOCtl interface for DOS mode applications is INT 21H.

See the *OS/2 Version 1.2 Programming Reference* for a detailed description of the DosDevIOCtl function call. See Chapter 7, "Generic IOCtl Commands," in this book for a detailed description of the IOCtl interface for device drivers.

# ASYNC (RS232-C) Communications Device Driver

The Asynchronous Communications (ASYNC) device driver enables OS/2 applications to utilize the Serial Communications (RS232-C) device hardware in both the IBM Personal Computer AT and the IBM Personal System/2 (PS/2)™ systems. The device driver will allow an application program in the OS/2 mode to support full duplex communications while the device driver:

- Services the RS232-C port in an interrupt-driven manner.
- Provides transmit and receive queues.
- Provides different automatic control modes of the modem control signals.
- Provides logical data stream flow control (XON/XOFF) for transmit and receive.

The user will normally want to use the ASYNC device driver either in conjunction with the Print Spooler (for serial printers only), or with an application program that utilizes the RS232 enabling capabilities of the ASYNC device driver coupled with a serial device attached to the system.

The ASYNC device driver can be installed optionally by the user by using a DEVICE= command in CONFIG.SYS. Because the system resources utilized by this device driver restrict memory available to DOS mode applications, this device driver should not be installed unless required.

Refer to "Category 1 ASYNC (RS232-C) Control IOCtl Commands" on page 7-5 for a detailed description of the Category 1 ASYNC Control IOCtl Commands referenced below.

## Hardware Support

Supported hardware for the RS232-C ASYNC communications device driver includes:

- The IBM Personal Computer AT, the IBM Personal Computer XT Model 286, and the IBM Personal System/2 Model 30 - 286. The IBM Personal Computer AT Serial/Parallel Adapter card (#0215, #3395, and #3400) is the supported adapter.
- The IBM Personal System/2 family computers (Models 50, 60, 70 and 80).

On some IBM ASYNC adapters and some IBM PS/2 system planar boards, the serial device provides Extended Hardware Buffering (also known as *FIFO mode operation*). This hardware feature is particularly helpful when performing multiple-character reads and/or writes from/to the COM ports. The ASYNC device driver will determine availability of this hardware feature during system initialization. This feature will be enabled only on those systems where the serial device hardware correctly supports FIFO mode operation.

## IBM Personal Computer AT Adapter Support

The device driver for the IBM Personal Computer AT supports a maximum of two ASYNC ports, each on separate interrupt levels. The device driver will recognize ASYNC ports with the following base I/O addresses:

- 3F8H (must generate a level 4 interrupt)
- 2F8H (must generate a level 3 interrupt).

No other base I/O addresses will be recognized or supported, and no other interrupt level combinations are supported.

The ASYNC device driver for the IBM Personal Computer AT supports:

- the IBM Personal Computer AT Serial/Parallel Adapter, based on the NS 16450 UART (Universal Asynchronous Receiver Transmitter) device; and
- other compatible adapters based on the INS 8250 UART architecture (NS 16550, NS 16550A).

The ASYNC device driver for the IBM Personal Computer AT interfaces directly to the hardware.

## IBM PS/2 Adapter Support

The ASYNC device driver for the IBM PS/2 uses the functions provided by the Advanced BIOS (ABIOS) interface for the ASYNC device. It does not interface directly to the hardware and does not support any device which requires direct hardware access.

The device driver supports a maximum of three ASYNC ports on a maximum of two different interrupt levels. The interrupt levels must have ABIOS support, with one unit per logical ID (LID) for the ASYNC device ID.

The only ASYNC devices supported on IBM PS/2 are COM1, COM2, and COM3. These devices correspond to the first three LIDs in the ABIOS common data area that have the architected ASYNC device ID. These devices also correspond to the first three ASYNC addresses in the ROM BIOS 40: data area.

If a device has capabilities other than ASYNC which cannot be utilized independently (for example, as in the Advanced BIOS separate LID architecture) of the ASYNC capabilities, and if Advanced BIOS assigns that device the ASYNC device ID, then that device can only be used for ASYNC in that power-on session.

If the device is not assigned the ASYNC device ID, it is not supported by this device driver.

If the device is assigned the ASYNC device ID and it has additional capabilities (for example, a built-in modem) beyond supporting the RS232-C port, the device driver will not recognize those additional capabilities (and potential limitations). Also, the device driver will not inform any application program of those additional capabilities (or limitations). And the device driver will not limit the control of the RS232-C interface or the device to only those modes which are acceptable to the extended hardware capabilities of that RS232-C port.

If an ASYNC device is not supported by OS/2 but is recognized by Advanced BIOS as an ASYNC device ID, the device driver may recognize and try to use that unsupported device if it is COM1, COM2, or COM3.

## Attachment Support

The ASYNC device driver does not provide any support for devices attached to the RS232-C port. The device driver provides enabling support for the RS232-C interface itself. Application programs, subsystems, and systems programs provide the support needed to use devices attached to the RS232-C port.

The ability to support a device can be determined by understanding the level of RS232-C interface enabling support the device driver provides, along with the characteristics of the attachment hardware in question and the required functions to be supported.

OS/2 provides a mechanism where one or more additional drivers can be installed to support specific COM ports. This feature may be required for the following reasons:

- To allow an application program to support a special device not adequately supportable with this ASYNC device driver.
- To allow additional COM ports (besides COM1-3 on IBM PS/2) to be supported.
- To enhance the level of device driver function for a given COM port. (This may be required for certain subsystem support.)

## RS232-C Interface

The ASYNC interface consists of separate read and transmit lines.

There are two separate modem control signals whose output values can be controlled by the device driver:

- Data Terminal Ready (DTR)
- Request To Send (RTS).

There are four separate modem control signals whose input values are available to the device driver:

- Data Set Ready (DSR)
- Clear To Send (CTS)
- Data Carrier Detect (DCD), also known as Receive Line Signal Detect (RLSD)
- Ring Indicator (RI).

The receive and transmit data lines have the following hardware characteristics:

- Logical 1 - Marking - More negative than -3 Volts. This state could mean no data.
- Logical 0 - Spacing - More positive than +3 Volts. This state could mean break condition.

The modem control signal lines have the following hardware characteristics:

- Function ON when more positive than +3 Volts.
- Function OFF when more negative than -3 Volts.

## Hardware Support for Extended Hardware Buffering

This capability is a feature of the asynchronous communications port's serial controller device. Serial controllers which support this capability (such as the NS-16550A UART) are present in many IBM PS/2 systems and on a variety of IBM and non-IBM asynchronous communications adapters.

The message "COMx supports Extended Hardware Buffering" is displayed during system initialization for each installed COM port (X=1, 2, or3) which supports Extended Hardware Buffering.

## INS 8250, INS 8250-B Considerations

The following hardware defects cannot be compensated for in the device driver and can cause indeterminate function when used with the OS/2 ASYNC Device Driver:

- Line Control Configurations

  These devices are known to transmit bad data when configured for 5 Data Bits and 1.5 Stop Bits.

- Receive Character Overrun Errors

  These devices are know to occasionally drop received characters without posting the RECEIVE OVERRUN error flag. Undetected data loss can result from this hardware deficiency. Application error-correction routines may be implemented to ensure accurate data transmission when these devices are being used.

- Spurious Characters at Power-On

  These devices can transmit a SINGLE random character at Power-On. The connected device should not be expecting valid data to be received until after the device driver initialization message has been displayed.

## Supported Bit (Baud) Rates on 16450 and Compatibles

The NS 16450 and other compatible UART devices (including the 8250- and 16550-Series UARTs) incorporate a Programmable Baud Generator feature that is driven as a function of the following constants:

- CLOCK = 1843200 ; crystal frequency
- CLOCK/16 = 115200 ; after divider

Given these constants, the algorithm for determining which rates are supported is explained in the following examples:

**Note:** Where division is performed and the quotient is not a whole integer, an integer result is obtained by ROUNDING OFF the fractional part of the quotient.

- If 900 baud is specified, the baud rate will be exactly 900 because it divides evenly into 115200 (115200/900 = 128). Bit rate returned by IOCTL function 61H is 900.
- If 901 baud is specified, the baud rate will not change and the IOCTL will fail with an invalid parameter error because it cannot be supported within .01% (115200/901 = 128, 115200/128 = 900, gives a .1111% error).
- If 907 baud is specified, the baud rate will be 907.0866 because it can be supported within .01% (115200/907 = 127, 115200/127 = 907.0866, gives a .0095% error). Bit rate returned by IOCTL function 61H is 907.
- If 110 baud is specified, the baud rate will be 110.0287, even though the error is over .01% (115200/110 = 1047, 115200/1047 = 110.0287, gives a .0260% error). Bit rate returned by IOCTL function 61H is 110.

# ASYNC (RS232-C) Device Driver Features

The device driver supports the ASYNC interface in an interrupt-driven manner. This allows the multi-tasking capabilities of OS/2 to be supported while ASYNC data reception and transmission is taking place.

**Note:** With any supported hardware, the device driver cannot absolutely guarantee accurate function, as there is a dependency on the hardware being driven. It is known, for example, that INS 8250 and INS 8250-B UART devices exhibit a number of deviations from their hardware specifications. In some cases, these deviations have been compensated for in the device driver design. Some of these deviations, however, cannot be resolved in software. The user should be familiar with the limitations and restrictions associated with such hardware. See "Special Hardware Considerations" for more information.

With the current ASYNC hardware, when data is given to the transmit hardware, data will be transmitted at the physical RS232-C interface. When data is given to the transmit hardware, it has not yet been physically transmitted (at the RS232 interface). The data is considered completely transmitted by the transmit hardware at the physical RS232 interface when the transmit shift register of the UART is empty. The IOCtl Return Transmit Data Status (Category 1 Function 65H) can be used to determine this information.

The device driver transmit queue is a memory buffer between OS/2 and the transmit hardware. It is considered to be owned by the device driver because the device driver controls the data movement into and out of the transmit queue. Algorithms for this data movement may change between releases of the device driver. Changes in the ASYNC hardware may cause changes in the data movement algorithms and/or external interfaces.

The device driver receive queue is a memory buffer between OS/2 and the receive hardware. It is considered to be owned by the device driver because the device driver controls the data movement into and out of the receive queue. Algorithms for this data movement also can change between releases of the device driver. Changes in the ASYNC hardware can cause changes in the data movement algorithms and/or external interfaces.

Data that applications send (made available by Write requests) get placed in the device driver transmit queue. When an interrupt occurs to tell the device driver that the hardware is ready for more data, the device driver will give the transmit hardware more data from the transmit queue.

When an interrupt occurs to tell the device driver that the hardware has received data, that data is placed in the device driver receive queue. When the device driver gets a read request (Read request packet) from the application, the device driver fills the read request from the receive queue.

At high baud rates, such as 19200 bits per second, a serial device supporting full-duplex asynchronous I/O may generate an interrupt every 260 microseconds (at 10 bits per character and one interrupt per character transmitted and received). This leads to excessive interrupt-time overhead in the multitasking, interrupt-driven, device driver.

To address this problem, serial devices with Extended Hardware Buffering (FIFO or First-In-First-Out buffers) capabilities have been developed. Many serially attached devices, however, which support the RS232-C interface, have been designed to operate with specific protocols that assume the system will process all data I/O one character at a time. The ASYNC device driver employs a software mechanism that automatically controls parameters to utilize the Extended Hardware Buffering capability, while compatibly supporting devices that use existing ASYNC device driver protocols.

The Automatic Protocol Override (system default) mode for Extended Hardware Buffering support will only partially utilize these performance advantages, while remaining fully compatible with the behavior of existing ASYNC device driver protocols (for example, Input Sensitivity using DSR). Applications and/or subsystems may disable certain device driver default settings in order to fully utilize the Extended Hardware Buffering capabilities. This will result in a significant reduction of serial device interrupt processing overhead, and will greatly increase the aggregate baud rates that can be supported across multiple active COM ports. See the section on Automatic Protocol Override for a more detailed description.

The size of the receive and transmit queues are available from the following IOCtls:

- Return number of chars in receive queue
  (Category 1 Function 68H)
- Return number of chars in transmit queue
  (Category 1 Function 69H).

The device driver services each communications port independently. Requests issued to a given port have no effect on any other communications ports that the device driver may be servicing.

The device driver processes Read and Write request packets independently for a given port. An application can be written to support simultaneous reception and transmission of data. In addition, the device driver can process an IOCtl request simultaneously with outstanding Read and Write requests.

The device driver does not schedule the processing of IOCtl requests. The device driver processes the IOCtl request when received, regardless of what else it is doing. This may cause unexpected results if, for instance, the baud rate is modified while data reception or transmission is taking place.

The application should issue only one IOCtl request at a time. If the application issues another IOCtl request before the first IOCtl request completes, the results are UNDEFINED.

The device driver will queue multiple Read and Write request packets independently. The device driver always will begin processing the Read request packets in the order that they are received. It will always begin processing the Write request packets in the order that they are received.

**Note:** The operating system does not guarantee that file system requests will be delivered to a device driver in the order in which they are issued by an application. This means that a request by one thread can get blocked in the operating system, thus allowing a subsequent request by a different thread for the same function (for example, DosWrite) to pass through and arrive ahead of the first thread at the device driver. This is true for synchronous operations performed by multiple threads, or asynchronous operations performed by the same thread.

Because of thread priority considerations and the system dynamics, the order observed by the application of completing requests of the same type may not be in the order that they were received by the device driver. The device driver will always keep the data in the same order in which the Read and Write request packets (of the same type) were received in. There is no ordering or timing between different types of request packets.

The concept of a *First Level Open* is described in the section on "States of the ASYNC Device Driver" on page 6-8. A First Level Open occurs when the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close). A CLOSE request packet causing the device driver to process the next OPEN request packet as a First Level Open, is called a Last Level Close. Because the requests that an application issues sometimes get out of order before they reach the device driver, an application cannot consider a CLOSE a Last Level Close until the CLOSE completes. If the application issues an OPEN request to the COM port before a previously issued CLOSE request is completed, then the results are UNDEFINED.

The Flush request may be completed before all the appropriate request packets (that have been queued by the device driver) have been flushed. The appropriate request packets will eventually be flushed (and return to the caller) based on their priority and the system dynamics. Once the Flush request has been processed, the appropriate request packets will not cause data to be transmitted (or received data to be moved) incorrectly.

The device driver supports different time-out processing characteristics and time-out settings for the Read and Write requests.

The device driver is the only one aware of when a given character is being transmitted or received at the hardware interface. Therefore, an application cannot expect to provide real-time flow control of data (in the middle of data transmission or reception) based on logical characters (XON/XOFF) or based on the state of the modem control signals by:

- Manually controlling or monitoring those modem control signals
- Manually monitoring the queue status
- Manually monitoring data moving across the link.

Alternatively, the device driver provides optional modes of operation to control the data flow through the RS232-C port automatically. OS/2 applications select which potocol(s) to be made active using IOCtls.

## Output Modem Control Signals

In addition to allowing the application to control RTS and DTR directly, the device driver has different automatic control modes to control the value of the output modem control signals. They are:

- Open and Close processing of DTR and RTS
- Disable/Enable DTR and RTS
- RTS toggling on transmit
- Input handshaking using DTR and RTS.

These different control modes are described in the section on "States of the ASYNC Device Driver" on page 6-8 and in the IOCtls description.

**Note:** The level of support provided by this device driver requires that DTR and RTS are turned on at least once, even if this puts the device driver in a mode where they will never be turned on again.

## Input Modem Control Signals

Besides allowing the application to read directly the current state of DSR, CTS, DCD, and RI, the device driver has different automatic modes that cause it to respond to the value that some input modem control signals may have. They are:

- Output handshaking using CTS, DSR, DCD
- Input sensitivity using DSR.

These different control modes are described in the section on "States of the ASYNC Device Driver" on page 6-8 and in the IOCtls description.

Additional information on the state of the input modem control signals is available by using the IOCtl Return COM Event Information (Category 1 Function 72H).

## Logical Flow Control (XON/XOFF)

The application can attempt to manually control the flow of data by using the following IOCtls:

- Transmit Immediate (Category 1 Function 44H)
- Stop Transmit behave as if XOFF received (Category 1 Function 47H)
- Start Transmit behave as if XON received (Category 1 Function 48H).

The device driver also will control automatically the flow of transmitted data based upon the reception of XON/XOFF characters. This is referred to as automatic transmit flow control (XON/XOFF) and is described in the section on "States of the ASYNC Device Driver" on page 6-8.

The device driver also will attempt to control the flow of data that is received by automatically transmitting XON/XOFF characters to the system it is communicating with, based on the amount of space left in the receive queue. This is referred to as automatic receive flow control (XON/XOFF) and is described in the section on "States of the ASYNC Device Driver" on page 6-8.

## Support for Extended Hardware Buffering

Another significant feature of this device driver is its exploitation of the Extended Hardware Buffering capabilities of the serial communications devices in many IBM systems and option adapters. Extended Hardware Buffering refers to the ability of the serial device servicing a COM port to buffer in hardware several characters and release them all at one time on the occurrence of a single transmit or receive hardware interrupt. This capability significantly reduces the interrupt-driven I/O processing overhead required to service Transmit and Receive requests on a given COM port. On the devices that support the Extended Hardware Buffering capability, this significantly improves COM I/O throughput and improves data integrity for higher data transfer rates (up to 19200 bits per second).

The Extended Hardware Buffering capabilities will be automatically controlled under the default modes of the ASYNC device driver. Automatic Protocol Override is a feature of the OS/2 ASYNC device driver that automatically controls parameters relating to Extended Hardware Buffering. Systems and Adapters that incorporate the "FIFO mode" hardware feature in a manner fully compatible with the NS-16550A UART will be automatically enabled to run in Automatic Protocol Override mode.

## Line Characteristics

IOCtls can be used to control and read the baud rate, number of stop bits per character, number of data bits per character, and the parity characteristics of the line. See the section on "States of the ASYNC Device Driver" on page 6-8.

## Break and Error Processing

The device driver can be commanded to transmit a Break with an IOCtl (Category 1 Function 4BH and Function 45H).

An application can detect where an error or break occurred in the input data stream by using Break Replacement Character Processing and Error Replacement Character Processing. This requires certain binary byte combinations to be reserved for this purpose. See the section on "States of the ASYNC Device Driver" on page 6-8.

## State of the COM Port

The following IOCtls can be used to determine the state of the COM port or if a given event happened. However, the exact timing relationship between this information and the specific data being received or transmitted at the time of the event is not available.

- Return COM Event Information (Category 1 Function 72H)
- Return COM Status (Category 1 Function 64H)
- Return COM Error (Category 1 Function 6DH).

### Event Notification

The device driver does not provide any capabilities of event notification. For example, the only way for an application to know that RI changed state or that a Break condition occurred is to poll that status with the IOCtl Return COM Event Information. This should not be a problem for those applications that can use the automatic control modes of the device driver during the course of a communications dialog (for time-critical control functions). Polling could be adequate for non-time-critical event monitoring.

## Error Alert Generation

The ASYNC device driver supports SNA Generic Alerts by generating Error Alerts, as defined under OS/2 Logging Facility. Alerts are generated by the ASYNC device driver whenever the OS/2 Logging Facility is enabled by the user at system initialization time.

Alerts may be generated only while the COM port is open and is processing a Write request (transmitting data). Write Timeout mode must be NORMAL. (If Infinite Timeout mode is enabled, timeouts do not occur.) Error Alerts can be generated only when a Write Timeout occurs while waiting for:

1. CTS to be asserted, when Transmit is disabled due to CTS being inactive.

   Output Handshaking Using CTS mode must be enabled for this alert generating condition to occur. This mode is enabled by default in the ASYNC device driver.

2. DSR to be asserted, when Transmit is disabled due to DSR being inactive.

   Output Handshaking Using DSR mode must be enabled for this alert generating condition to occur. This mode is enabled by default in the ASYNC device driver.

3. DCD to be asserted, when Transmit is disabled due to DCD being inactive.

   Output Handshaking Using DCD mode must be enabled for this alert generating condition to occur. This mode must be enabled by an application. It is not enabled by default in the ASYNC device driver.

4. An XON to be received, when Transmit is disabled due to an XOFF being received.

   Automatic Transmit Flow Control mode must be enabled for this alert generating condition to occur. This mode must be enabled by an application. It is not enabled by default in the ASYNC device driver.

See "Write Timeout State" in the next section on page 6-15 for more information on events that can cause these conditions to occur. See also IOCtl function 53H, category 1, Set Device Control Block (DCB) Parameters note 8 in chapter 7.

## States of the ASYNC Device Driver

This section itemizes the different processing states of the ASYNC device driver, the ASYNC hardware, and the ASYNC control signals.

The items that will be discussed are:

- Baud Rate
- Data Bits
- Parity
- Stop Bits
- DTR and RTS
- DTR Control Mode
- RTS Control Mode
- Transmitting Break
- COM Event Word and COM Error Word
- Output handshaking using CTS, DSR, DCD
- Input sensitivity using DSR
- Automatic Transmit Flow Control (XON/XOFF)
- Automatic Receive Flow Control (XON/XOFF)
- XON/XOFF Characters
- Error Replacement Character Processing
- Error Replacement Character
- Break Replacement Character Processing

- Break Replacement Character
- Null Stripping
- Write Time-out State
- Write Time-out Value
- Read Time-out State
- Read Time-out Value
- Transmit Immediate
- Extended Hardware Buffering.

The following will be given for each item:

- A brief description
- The initial (default) value
- The effect on the device driver when the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close). This is called a First Level Open. If applicable, the way the state of the device driver is affected by a close request packet.
- How the MODE utility (in OS/2 mode) can be used to alter the state of this item or how the MODE utility (in OS/2 mode) will alter the state of this item.
- The effect on the device driver of the state of Extended Hardware Buffering. In particular, special considerations relating to Automatic Protocol Override are given where the protocol being described is affected by this feature of the ASYNC device driver.

## Baud Rate

Baud rate determines the hardware setting for the data transfer rate, specified in bits per second. Baud rate is the speed for which the hardware is configured. See IOCtls "Set Baud Rate" (Category 1 Function 41H) and "Return Current Baud Rate" (Category 1 Function 61H).

Initial Value - 1200 baud.

First Level Open - No effect.

Mode Utility - User interface to change the baud rate.

## Data Bits

The number of bits that are contained in each character transmitted or received by way of the communications hardware. See IOCtls "Set Line Characteristics" (Category 1 Function 42H) and "Return Line Characteristics" (Category 1 Function 62H).

Initial Value - 7 data bits.

First Level Open - No effect.

Mode Utility - User interface to change the number of data bits.

## Parity

Determines whether a parity bit exists and (if appropriate) what algorithm determines its value. See IOCtls "Set Line Characteristics" (Category 1 Function 42H) and "Return Line Characteristics" (Category 1 Function 62H).

Initial Value - Even parity.

First Level Open - No effect.

Mode Utility - User interface to change the parity characteristics.

## Stop Bits

Determines the number of stop bits associated with each character transmitted or received by way of the communications hardware. See IOCtls "Set Line Characteristics" (Category 1 Function 42H) and "Return Line Characteristics" (Category 1 Function 62H).

Initial Value - 1 stop bit.

First Level Open - No effect.

Mode Utility - User interface to change the number of stop bits.

## DTR and RTS

The value of the modem control signals Data Terminal Ready (DTR) and Request To Send (RTS) put out by the communications hardware. Each signal is controlled independently and can be either ON or OFF. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Return Modem Control Output Signals" (Category 1 Function 66H).

Initial Value - When the device driver starts the port during device driver initialization, their values will be turned off.

First Level Open - The signals are normally turned on but there are many conditions that may cause the signals to be affected differently. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Set Device Control Block" Information NOTE 1 (Category 1 Function 53H) for a complete explanation.

Close Considerations - A close request packet, when after processing this close request the port will not be open any more from another open without a close (last level close), will cause DTR and RTS to be turned OFF after the transmit hardware has completely transmitted all the data that it has been given to transmit by the device driver AND at least 10 additional character times have elapsed (or one second, whichever is less).

Mode Utility - Not applicable for direct control. Indirect effects through altering processing modes of the device driver are possible.

## DTR Control Mode

The different control modes for DTR are:

- Enable
- Disable
- Input Handshaking.

The Enable and Disable control modes of DTR affect DTR processing during a First Level Open. When these control modes are set through the Category 1 Function 53H IOCtl, the value of the DTR signal may be modified immediately by the device driver. The action will depend on the previous control mode of DTR and the current value of the DTR modem control signal. If the control mode of DTR is Input Handshaking, then the device driver will control the DTR signal, depending on how full the receive queue is. The bits that control these states of the device driver are in the device control block. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Set Device Control Block Information" NOTE 1 (Category 1 Function 53H).

Initial Value - Enable.

First Level Open - No effect.

Mode Utility - User interface to change the DTR Control Mode of the device driver.

## RTS Control Mode

The different control modes for RTS are:

- Enable
- Disable
- Input Handshaking
- Toggling on Transmit.

The Enable and Disable control modes affect RTS processing during a First Level Open. When these control modes are set using the Category 1 Function 53H IOCtl, the value of the RTS signal may be immediately modified by the device driver. The action will depend on the previous control mode of RTS and the current value of the RTS modem control signal. If the control mode of RTS is Input Handshaking, the device driver will control the RTS signal, depending on how full the receive queue is. If the control mode of RTS is Toggling on Transmit then the device driver will control the RTS signal, depending on whether it is transmitting data. The bits that control these states of the device driver are in the device control block. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Set Device Control Block Information" NOTE 1 (Category 1 Function 53H).

Initial Value - Enable.

First Level Open - No effect.

Mode Utility - User interface to change the RTS Control Mode of the device driver.

## Transmitting Break

The device driver can be transmitting a break. See IOCtls Break On (Category 1 Function 4BH) and Break Off (Category 1 Function 45H).

Initial Value - Not transmitting a break.

Close Considerations - A CLOSE request packet, when after processing this close request the port will not be open any more from another open without a close (last level close), will cause the device driver to tell the hardware not to transmit a break any more.

Mode Utility - Not applicable.

## COM Event Word and COM Error Word

These two words have bits which show status of the COM port. When an event happens the appropriate bits are turned on. The bits are cleared when the word is read with the appropriate IOCtl. See IOCtl Return COM Event Information (Category 1 Function 72H) and Return COM Error (Category 1 Function 6DH).

Initial Value - All defined bits are 0.

First Level Open - All defined bits are 0.

Mode Utility - Not applicable.

## Output Handshaking using CTS, DSR, DCD

This mode of the device driver can be controlled independently for each modem control signal. When this mode of the device driver is enabled, the device driver will not give data to the transmit hardware if the appropriate modem control signal is OFF. See IOCtl "Set Device Control Block Information" NOTE 3 (Category 1 Function 53H).

If one of these modes is enabled, Error Alerts may be generated when the OS/2 Logging Facility is enabled. If an external device causes CTS, DCD, and/or DSR to become inactive, transmit will be blocked waiting for the respective line to become active again. If the line does not become active before the Write Timeout period expires, an Error Alert is generated.

Initial Value - Output handshaking using CTS and DSR is enabled. Output handshaking using DCD is disabled.

First Level Open - No effect.

Mode Utility - User interface to enable/disable this mode of the device driver for CTS and DSR (independently). Mode will always disable this mode of operation of the device driver for DCD.

Automatic Protocol Override Considerations - When Output Handshaking using DSR (Data Set Ready), CTS (Clear To Send), or DCD (Data Carrier Detect) is enabled, the device driver will respond to the dropping modem line within a single character time. That is, Automatic Protocol Override controls the Extended Hardware Buffering capability such that only one character at a time is given to the transmit hardware (Transmit Buffer Load Count is set to 1). The Receive Trigger Level Level is unaffected.

## Input Sensitivity Using DSR

When the device driver is enabled for this mode of operation and DSR is OFF, the device driver will discard receive data. See IOCtl "Set Device Control Block Information" NOTE 4 (Category 1 Function 53H).

Initial Value - Input Sensitivity using DSR is enabled.

First Level Open - No effect.

Mode Utility - User interface to enable/disable this mode of the device driver.

Automatic Protocol Override Considerations - When Input Sensitivity Using DSR is enabled, the device driver will respond to the lowering of the DSR line within a single character time. That is, the Extended Hardware Buffering (Receive Trigger Level) will be adjusted so that the device will generate an interrupt for each character received. The full 16-character receive buffer is still available to help avoid any receive hardware overruns. The Transmit FIFO feature is also still enabled so that only one transmit interrupt will occur for every 16 characters transmitted.

**Note:** In situations where DSR may naturally drop from ON to OFF at the end of a communications session, but is not normally toggled during the session, it will be most advantageous to DISABLE Input Sensitivity Using DSR after the communications data transfer has begun. This achieves a significant performance benefit (under the control of Automatic Protocol Override), without the risk of losing valid data upon termination of the session. If, however, the DSR signal is expected to toggle frequently during a communications session, Input Sensitivity Using DSR should not be disabled, or potential data loss may result.

## Automatic Transmit Flow Control (XON/XOFF)

When the device driver is enabled for this mode of operation, the device driver will stop sending data to the transmit hardware when an XOFF is received, and resume sending data to the transmit hardware when an XON is received. See IOCtl "Set Device Control Block Information" NOTE 2 (Category 1 Function 53H).

If this mode is enabled, Error Alerts may be generated when the OS/2 Logging Facility is enabled. If an external device sends an XOFF but does not send an XON, transmit will be blocked waiting for the XON to be received. If the XON is not received before the Write Timeout period expires, an Error Alert is generated.

Initial Value - Automatic transmit flow control is disabled.

First Level Open - No effect on whether the device driver is enabled or disabled for this mode of operation. The state of the device driver will be reset to show that it has not received an XOFF so it can transmit (due to automatic transmit flow control) if it is enabled for this mode of operation.

Mode Utility - User interface to enable/disable this mode of the device driver.

Automatic Override Considerations - When Automatic Transmit Flow Control is enabled, the device driver will respond to receiving the XOFF within a single character time. That is, Automatic Protocol Override controls the Extended Hardware Buffering capability such that only one character is buffered for transmit at a time, and the device will generate an interrupt for every character received (Receive Trigger Level is set to 1).

## Automatic Receive Flow Control (XON/XOFF)

When the device driver is enabled for this mode of operation, the device driver will transmit an XOFF when its receive queue gets close to full, and an XON when its receive queue is about half full.

The NORMAL mode of Automatic Receive Flow Control causes the ASYNC device driver to stop transmitting all data after it sends an XOFF character until its receive queue lowers to about half full, when it sends the XON character. This behavior is suitable for most devices, but reduces transmit throughput significantly in Full-Duplex (Transmit and Receive) communications scenarios. By setting the Full-Duplex mode of Automatic Receive Flow Control, application data will continue to be sent by the ASYNC device driver after it sends the XOFF character. See IOCtl "Set Device Control Block Information" NOTE 2 (Category 1 Function 53H).

Initial Value - Automatic Receive Flow Control is disabled.

First Level Open - No effect on whether the device driver is enabled or disabled for this mode of operation. The state of the device driver will be reset to show that the last flow control character automatically transmitted was an XON if it is enabled for this mode of operation.

Close Considerations - If the last automatically transmitted character by the device driver was an XOFF and a CLOSE request packet is received, (when after processing this close request the port will not be open any more from another open without a close - Last Level Close), the device driver will automatically transmit an XON, if possible.

Mode Utility - Always disables Automatic Receive Flow Control.

### XON/XOFF characters

The characters used for automatic transmit and automatic receive flow control. See IOCtl "Set Device Control Block Information" NOTE 2 (Category 1 Function 53H).

Initial Value - XON is 11H and XOFF is 13H.

First Level Open - The XON and XOFF characters are reset to their initial values.

Mode Utility - No effect.

### Error Replacement Character Processing

The processing that the device driver performs when a received character had an error (parity, framing, overrun, or lack of receive queue space) is determined by whether error replacement character processing is enabled (active). See IOCtl "Set Device Control Block Information" NOTE 5 (Category 1 Function 53H).

Initial Value - Error replacement character processing is disabled.

First Level Open - Error replacement character processing is disabled.

Mode Utility - No effect.

### Error Replacement Character

The character value that the device driver uses if Error Replacement Character Processing is enabled. See IOCtl "Set Device Control Block Information" NOTE 5 (Category 1 Function 53H).

Initial Value - 00H.

First Level Open - Reset to 00H.

Mode Utility - No effect.

### Break Replacement Character Processing

If break replacement character processing is enabled, and the device driver detects a break condition, it will place the break replacement character in the device driver receive queue. If break replacement character processing is disabled, the device driver will not place any character in the device driver receive queue when it detects a break condition. See IOCtl "Set Device Control Block Information" NOTE 7 (Category 1 Function 53H).

Initial Value - Break replacement character processing is disabled.

First Level Open - Break replacement character processing is disabled.

Mode Utility - No effect.

### Break Replacement Character

The character value that the device driver uses if Break Replacement Character Processing is enabled. See IOCtl "Set Device Control Block Information" NOTE 7 (Category 1 Function 53H).

Initial Value - 00H.

First Level Open - Reset to 00H.

Mode Utility - No effect.

### Null Stripping

If the device driver is enabled for null stripping, characters read in from the receive hardware (non error or non break) with a value of 00H are thrown away. These null characters are stripped (not checked for Automatic Transmit Flow Control) even if the XON or XOFF character has been set to 00H, and are not placed in the device driver receive queue. See IOCtl "Set Device Control Block Information" NOTE 6 (Category 1 Function 53H).

Initial Value - Null stripping is disabled.

First Level Open - Null stripping is disabled.

Mode Utility - No effect.

## Write Time-out State

When the device driver processes a WRITE request packet, it can be with normal or infinite time-out processing. With normal time-out processing, if no data is given to the transmit hardware within a specified amount of time, the request will be completed. With infinite time-out processing, the request will be completed only when all the data from the request has been given to the transmit hardware. See IOCtl "Set Device Control Block Information" NOTE 8 (Category 1 Function 53H).

Error Alerts may be generated on the occurrence of Write Timeout if the OS/2 Logging Facility is enabled. In order for an error alert to be generated by the ASYNC device driver, the appropriate mode of operation must be enabled. See Error Alert Generation on page 6-8, as well as "Automatic Transmit Flow Control" and "Output Handshaking Using CTS, DSR, DCD".

Initial Value - Normal Write time-out processing.

First Level Open - No effect on write time-out processing.

Mode Utility - User interface to set infinite or normal write time-out processing.

## Write Time-out Value

The user specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the write time-out processing, if normal write time-out processing is enabled. See IOCtl "Set Device Control Block Information" NOTE 8 (Category 1 Function 53H).

Initial Value - 1 Minute.

First Level Open - Set to 1 Minute.

Mode Utility - No effect.

## Read Time-out State

When the device driver processes a READ request packet, it can be with normal, with NO-WAIT, or with Wait For Something time-out processing. With normal time-out processing, if no data is received in the specified period of time, the request will be completed. With NO-WAIT time-out processing, the request will obtain whatever data is available in the device driver receive queue (at the time the request is processed by the device driver) and return. With Wait-For-Something time-out processing, the request will act like NO-WAIT time-out processing. However, if no data is available when the device driver processes the request, the device driver will wait until some data is available or until the request times out due to normal time-out processing. See IOCtl "Set Device Control Block Information" NOTE 9 (Category 1 Function 53H).

Initial Value - Normal Read time-out processing.

First Level Open - The device driver is set to Normal Read Time-out processing.

Mode Utility - No effect.

## Read Time-out Value

The user specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the read time-out processing, if normal read time-out or Wait For Something time-out processing is enabled. See IOCtl "Set Device Control Block Information" NOTE 9 (Category 1 Function 53H).

Initial Value - 1 Minute.

First Level Open - Set to 1 Minute.

Mode Utility - No effect.

## Transmit Immediate

The device driver may be told to transmit a byte immediately, bypassing the normal file system write requests (bypassing the data to be transmitted in the transmit queue). Only one character at a time can be waiting to be transmitted "immediately." See IOCtl "Transmit This Byte Immediately" Category 1 Function 44H.

Initial Value - There is no character waiting to be transmitted immediately.

First Level Open - There is no character waiting to be transmitted immediately.

Close Considerations - A CLOSE request packet, when after processing this close request the port will not be open any more (from another open without a close), will cause the device driver to attempt to transmit the character waiting to be transmitted immediately. If the device driver cannot transmit the character waiting to be transmitted immediately (See IOCtl "Transmit this Byte Immediately" Category 1 Function 44H), then it will not try to transmit the character and will proceed with the close processing.

Mode Utility - Not applicable.

## Extended Hardware Buffering

The extended hardware buffering (FIFO-mode) capabilities available in supported systems applies to the NS-16550A UART device and other fully compatible devices. These serial devices are installed on many IBM PS/2 system planars and on various Asynchronous Communications adapter options. Refer to the "Hardware Support for Extended Hardware Buffering" section in this chapter.

On those systems which incorporate serial device(s) that fully and compatibly support Extended Hardware Buffering, the OS/2 ASYNC device driver provides three modes for exploiting this feature:

- ENABLED
- DISABLED
- Automatic Protocol Override.

The default will be to enable Automatic Protocol Override on that COM port. Automatic Protocol Override is an ASYNC Device Driver mode of operation that automatically controls various parameters of Extended Hardware Buffering, such as Receive Trigger Level and Transmit Buffer Load Count.

Automatic Protocol Override causes the Receive Trigger Level and Transmit Buffer Load Count to be adjusted according to the device driver handshaking protocols in effect. When Automatic Protocol Override mode is on and the handshaking protocols are set to their default settings, the device driver will only partially exploit the performance advantages of Extended Hardware Buffering. The default handshaking protocols are:

- Enabled for Input Sensitivity Using DSR
- Enable for Output Handshaking Using CTS and DSR
- Disable for Output Handshaking Using DCD
- Disabled for Automatic Transmit Flow Control.

If both Input Sensitivity Using DSR and Output Handshaking Using CTS and DSR are disabled, the Automatic Protocol Override will cause the ASYNC device driver to automatically reset internal parameters to fully exploit the Extended Hardware Buffering capabilities to the maximum extent possible.

The device driver's initialization default is to set Extended Hardware Buffering capabilities to the Automatic Protocol Override mode. An application or subsystem may alternatively set Extended Hardware Buffering DISABLED, which will cause the hardware to service transmit and receive interrupts one character at a time. It may also set Extended Hardware Buffering ENABLED, which will cause the device driver to set certain Extended Hardware Buffering parameters to specified levels, such that the serial device fully exploits its Extended Hardware Buffering capabilities to the maximum extent possible.

When Extended Hardware Buffering is set to ENABLED, the following serial device hardware capabilities will be exploited for maximum performance benefit and increased receive data integrity:

- By setting the Transmit Buffer Load Count to 16, up to 16 characters are placed in the transmit hardware buffer (FIFO) during the processing of one "Transmit Holding Register Empty" (THRE) interrupt.
- A 16 character receive hardware buffer (FIFO) is available with the Receive Trigger Level set to 1, 4, 8, or 14 characters. The Receive Trigger Level determines when the receive hardware will generate a "Receive Data Available" hardware interrupt.

If the device driver receives an OPEN request for a COM port that is not already open, it will not alter the Extended Hardware Buffering setting which is in effect at that time. The ASYNC device driver will preserve this state across multiple OPEN and CLOSE requests.

Initial Value - Automatic Protocol Override mode is on.

First Level Open - No effect

Mode Utility - User interface to set the Extended Hardware Buffering mode to ENABLED, DISABLED, or to Automatic Protocol Override.

## Reserved Device Names- COM1-N

The device name AUX does not appear in the Device Header of the ASYNC device driver. The ASYNC device driver does not support the reserved name AUX for either DOS mode applications or OS/2 mode applications.

### IBM Personal Computer AT Considerations- COM1, COM2

The IBM Personal Computer AT ASYNC device driver will have device names COM1 and COM2 in its Device Header.

Device name COM1 will correspond to the ASYNC hardware with its base address in 40:0 (during initialization) and device name COM2 will correspond to the ASYNC hardware with its base address in 40:2 (during initialization). After the ASYNC device driver initialization, the contents of the 40: area are not monitored by the ASYNC device driver. The values in the 40: area are set to 0 for any COM port that is initialized. If these values are restored later in the same power on session, the integrity of the ASYNC device driver could be adversely affected by any process that directly accesses hardware through the 40: area.

It should be noted that if the adapter card is configured as a secondary COM port, the system may not always treat it as COM2.

**Note:** The mapping of the 40: area to COMn must be consistent across all device drivers that support the ASYNC hardware in the IBM Personal Computer AT hardware environment.

Refer to the SETCOM40 Utility for more information.

### IBM PS/2 Considerations- COM1 - 3

The IBM PS/2 ASYNC device driver will have device names COM1, COM2, and COM3 in its Device Header.

Device name COM1 will correspond to the first LID in the Advanced BIOS common data area with the ASYNC device ID. Device name COM2 will correspond to the second LID and device name COM3 will correspond to the third LID in the Advanced BIOS common data area with the ASYNC device ID. The Advanced BIOS architecture ensures that the ordering in the ROM BIOS 40: data area will match the ordering of the LIDs in the common data area. Compatibility BIOS supports up to 4 ASYNC devices on IBM PS/2 and the device driver assumes that the order of the Logical IDs match the order of the addresses of these devices in the ROM BIOS 40: data area.

Additional ASYNC devices may be supported by additional device drivers.

The mapping of the ASYNC Logical ID ordering to COMn must be consistent across all device drivers that support the ASYNC hardware in the IBM PS/2 hardware environment.

## Initialization / Resource Management

The device driver is loaded and initialized with a DEVICE= statement in CONFIG.SYS. The name of the device driver for the IBM Personal Computer AT is COM01.SYS and the name of the device driver for the IBM PS/2 is COM02.SYS. The device driver does not process any parameters on the DEVICE= statement. It is the responsibility of the installation process or the user to have the correct DEVICE= statement in the CONFIG.SYS file, depending on whether OS/2 is installed on:

- IBM Personal Computer AT and IBM Personal Computer XT Model 286 - COM01.SYS
- IBM Personal System/2 Models 50, 60, 70, and 80 - COM02.SYS.

During initialization, the device driver will attempt to free memory from its data segment for ports it does not need that do not get initialized. The device driver will not remove device names from its Device Header for ports that do not get initialized.

The device driver will not deinstall a device if the system requests it. If another device driver wishes to support a port already supported by this device driver, it needs to initialize before this ASYNC device driver.

Shared ownership of a given serial device between multiple device drivers in a single boot of OS/2 is supported, subject to certain restrictions. Each device driver that installs sharing a serial device obtains exclusive access to that device when it processes an Open request, rather than claiming the device during initialization. It also fully relinquishes control of that device when it processes a matching Close request. See IBM Personal Computer AT Initialization Considerations, IBM PS/2 Initialization Considerations and File System Requests, for a description of the conventions pertaining to this co-existence.

When the device driver is initialized, it is enabled by default for Output Handshaking Using CTS and DSR. This is for compatibility with existing systems (IBM PC and PS/2 BIOS INT 14H) and the requirements of supporting an RS232 port. It is also enabled by default for Input Sensitivity to DSR. This allows a remote device to indicate whether data being received is valid, and is enabled to help ensure compatibility with existing systems. The initialization default for Extended Hardware Buffering on serial devices that fully support FIFO mode operations is Automatic Protocol Override.

**Note:** The ASYNC device driver default protocols provide an upwardly compatible RS232-C interface for communications with most devices. New communications applications and devices may not require the default protocols to be enabled. The user should evaluate these alternatives and consider which protocols to enable or disable in order to provide the highest possible performance for support of a given serial device.

## IBM Personal Computer AT Initialization Considerations

The device driver does not attempt to initialize or support a port if it does not get the INIT request packet for the port's corresponding device name. If the device driver gets the INIT request packet for a given device name, the device driver checks to see if a valid I/O address (2F8H or 3F8H) is in the BIOS 40: data area that corresponds to that device name. COM1 is in 40:0 and COM2 is in 40:2. If the 40: area does not have a valid I/O address, the device driver fails the initialization of this port and will not support this port. Otherwise, the device driver attempts to get exclusively the interrupt level that corresponds to the I/O address for this port. If the interrupt level is not available, the device driver fails the initialization of this port and will not support this port.

If the interrupt level is available, the device driver relinquishes the interrupt level and sets the 40: area address for this port to zero. This will normally prevent INT 14H from functioning for this port. The driver initializes the port, and sets up to support the port during this start-up of OS/2.

In summary, in order for the device driver to support a port on the IBM Personal Computer AT, the following must be true:

- The device driver must get an INIT request packet for the device name.
- The 40: area that corresponds to the device name must have a valid I/O address.
- The appropriate interrupt level must be available for exclusive use, even though the device driver will not claim the interrupt level for exclusive use during initialization.

The device driver claims ownership of the port by not DEINSTALLING the corresponding device name and by setting the corresponding 40: area to zero.

Another device driver can cause this device driver not to claim a port by initializing before this device driver and by doing at least one of the following:

- Not allowing this device driver to receive an INIT request packet for a given device name
- Putting an invalid I/O address in the corresponding 40: area (for example, 0)
- Owning exclusively the appropriate interrupt level at initialization time.

### IBM PS/2 Initialization Considerations

The device driver does not attempt to initialize or support a port if it does not get the INIT request packet for the port's corresponding device name. If the device driver gets the INIT request packet for a given device name, it attempts to claim ownership of the specific LID position for the ASYNC device ID that corresponds to the device name being initialized.

If the LID is not available, the device driver fails the initialization of this port and will not support this port.

If the LID is available, the device driver initializes the port and sets up support for this port during this start-up of OS/2. The device driver sets the appropriate port address in the BIOS 40: data area to zero (this will normally prevent INT 14H from functioning for this port), and the LID for this port is relinquished (it is reclaimed during Open processing).

In summary, for the device driver to support a port on IBM PS/2, the following must be true:

- The device driver must get an INIT request packet for the device name
- The ASYNC LID corresponding to the device name must be available.

The device driver claims ownership of the port by not DEINSTALLING the corresponding device name.

Another device driver can cause this device driver not to claim a port by initializing before this device driver and doing at least one of the following:

- Not allowing this device driver to receive an INIT request packet for a given device name
- Claiming the appropriate ASYNC LID.

## Data Translation / Monitor Support /Spooler Support

The device driver provides no data translation, code page, or monitor support. It is the responsibility of the application or subsystem to provide any function required in these areas.

Spooling from LPT to COM is supported by the print spooler, but spooling from COM to LPT or COM to COM is not supported. The level of code page support provided by the print spooler is available through LPT and the print spooler.

Any requests for registering or opening a monitor chain to COMn will be rejected by the device driver.

The device driver deals with binary data and provides no special processing in the area of "binary" or "ASCII" mode.

## ASYNC Communication Device Driver Interfaces

The ASYNC device driver supports the following File System requests:

| | |
|---|---|
| Init | Initialize the device |
| Open | Open the device |
| Close | Close the device |
| Read | Read from the device (Normal, Non-Destructive No Wait) |
| Write | Write to the device |
| Status | Input or Output status |
| Flush | Flush or terminate all pending requests |

In addition, the ASYNC device driver supports a large set of generic IOCtl functions (Category 1) that are organized into the following groups:

- Line Characteristics
- Manual XON/XOFF Processing
- Break Processing
- Device Driver I/O Queue Management
- Polled Event Functions
- Device Control Block (DCB) Parameter Access

## File System Requests

**Open Processing:** The device driver does not claim the interrupt level the port is on until the port is open. If the interrupt level is not available, the Open request packet is failed. The interrupt level is claimed exclusively on the IBM Personal Computer AT. The interrupt level is claimed shareable on the IBM PS/2.

On PS/2 machines, a first-level open causes the device driver to attempt to obtain a logical ID. If this fails (which indicates another device driver may be using the device), a general failure is returned. If the Logical ID is obtained, but the Open fails for some other reason, the Logical ID is freed. Other device drivers that co-exist with the OS/2 ASYNC device driver perform similar processing.

If a timer tick handler is not available during First Level Open processing, the open request may fail.

If the device driver receives an OPEN request packet and the COM device is not already open (from a previous open without a close), the device driver does special processing. See "States of the ASYNC Device Driver." This is called a First Level Open. If a subsequent open request is issued before a previous First Level Open request has completed, the device driver may process the open request packets in a different order than they were issued. This could cause the First Level Open to take effect at a time different from what the application expected.

An OPEN request should never be issued until a previous Last Level Close request has completed. If an OPEN request is issued before a previous Last Level CLOSE has completed, the function performed by a Last Level Close and a First Level Open may not occur.

If the port on the IBM Personal Computer AT is not already open (First Level Open), the device driver attempts to clear out any data in the receive hardware. On IBM PS/2, if the port is not already open (First Level Open), the device driver relies on the "reset / initialize" Advanced BIOS function to reset and clear the UART receive hardware.

**Close Processing:** An application should never close an open handle to a COM port while there are requests still pending for that handle. If a request has not completed, it may be waiting for time-out processing. IOCtls may be used to determine the current time-out processing and to change the current time-out processing.

A **Last Level Close** occurs when the port will not be open any more (from another open without a close). When a last Level Close occurs, the device driver will do some special processing. The device driver::

- Clears the receive and transmit queues
- Turns break off if currently transmitting a break
- Clears any character waiting to be transmitted immediately if it cannot be transmitted. If it can be transmitted, the device driver makes sure that it is given to the transmit hardware
- Attempts to automatically transmit an XON (if possible), if currently enabled for automatic receive flow control (XON/XOFF) and the last character that the device driver automatically transmitted was an XOFF
- Waits until all the data in the transmit hardware has been physically transmitted
- Unclaims the interrupt level
- Turns DTR and RTS OFF, if they are not already OFF. The device driver first waits the specified number of character times.
- Relinquishes the Logical ID for the device (on PS/2 machines).

(See "States of the ASYNC Device Driver" on page 6-8.)

**Read Processing:** The device driver begins processing Read requests in the order received by the device driver.

**Note:** This may not be the same order that the requests were issued by the application. If the device driver receives more than one Read request, the request is queued on the Read request queue for later processing.

Applications may not see Read requests completed in the order they were issued. The order of the data placed in the Read requests will reflect the order the requests were received by the device driver.

The data for the Read requests comes from the device driver receive queue. Because of time-out processing, it is normal for the total number of Read characters requested not to be read. This is not considered an error. The request is completed when time out is completed or when the amount of data requested is placed in the Read buffer.

The different kinds of Read Time-out processing are discussed in the "States of the ASYNC Device Driver" on page 6-8.

To reduce the probability of a device driver receive queue buffer overrun, the communications protocol should take into account the size of the device driver receive queue.

**Write Processing:** The device driver begins processing Write requests in the order received by the device driver.

**Note:** This may not be the same order that the requests were issued by the application. If the device driver receives more than one Write request, the request is queued on the Write request queue for later processing.

Applications may not see Write requests complete in the order they were issued. The order of the data transmitted due to the Write requests will reflect the order that the requests were received by the device driver.

The data from the Write requests are placed in the device driver transmit queue. The number of characters written are considered to be the number of characters given to the transmit hardware and not the number of characters placed in the device driver transmit queue. Because of time-out processing, it is possible that the total number of Write characters requested will not be transmitted. This is not considered an error. The request is completed when it times out or when the amount of data requested is given to the transmit hardware (but not actually transmitted at the physical RS232C interface).

The different kinds of Write time-out processing are discussed in the section on "States of the ASYNC Device Driver" on page 6-8.

If infinite Write time-out processing is enabled, it is the responsibility of the application to monitor the status of the Write requests. The application may have to issue an IOCtl to disable infinite Write time-out processing to cause the Write request to complete (without all the data being transmitted). If an application does not wish to check that all the data is given to the transmit hardware on each Write request, the application can use the infinite time-out processing mode of the device driver to ensure that all the data has been given to the transmit hardware before the request completes.

To increase the throughput (ratio of number of characters transmitted per second to the baud rate), the application should keep the Write requests as large as possible.

## Access Authorization

The ASYNC device driver does not prevent multiple processes from concurrently opening the same device name. The device driver uses standard file system contention control. An application or subsystem which needs exclusive access to the COM device should open it with access rights set to DENY_ANY. Inheritance of open handles and sharing of the device among multiple opens work in a manner similar to regular files.

## ASYNC (RS232-C) Generic IOCtl Command Summary

| Function | Description - Category 1 IOCtls |
|---|---|
| 49H | Reserved |
| | Line Characteristics |
| 41H | Set baud (bit) rates |
| 42H | Set line characteristics (stop, parity, data bits) |
| 46H | Set modem control signals |
| 61H | Return current baud (bit) rate |
| 62H | Return line characteristic (and break) |
| 66H | Return modem control output signals |
| 67H | Return current modem input signals |
| | Manual XON/XOFF (flow control) processing |
| 44H | Transmit immediate |
| 47H | Behave as if XOFF received (stop transmit) |
| 48H | Behave as if XON received (start transmit) |
| | Break Processing |
| 4BH | Break on |
| 45H | Break off |
| | Device Driver I/O Queue Management |
| 68H | Return number of characters in receive queue |
| 69H | Return number of characters in transmit queue |
| | Polled Events |
| 72H | Return COM event information |
| 64H | Return COM status |
| 65H | Return transmit data status |
| 6DH | Return COM error |
| | Device Control Block (DCB) Parameter Access |
| 53H | Set device control block parameters |
| 73H | Return device control block parameters |

Where device control block parameters determine:

- Automatic transmit flow control (start/stop transmit when XON/XOFF received)
- Automatic receive flow control (transmit XON/XOFF when receive buffer fills/empties).
- XON/XOFF characters
- DTR control mode (enable/disable/input handshaking)
- RTS control mode (enable/disable/input handshaking/toggling on transmit)
- Output handshaking using CTS/DSR/DCD (control signal determines when to transmit)
- Input sensitivity using DSR (reception of data controlled by DSR)
- Error replacement character and processing
- Break replacement character and processing
- Null stripping
- Timeout processing
- Extended Hardware Buffering (disabled/enabled/Auomatic Protocol Override).

See Chapter 7 for detailed descriptions of the ASYNC Category 1 IOCtl functions.

## DOS Mode Considerations / Restrictions

DOS mode applications that go directly to the COM hardware or that use INT 14H are not supported. The 40: area is set to 0 by the device driver for those ports serviced by the device driver.

Previous-level DOS device drivers that go directly to COMn hardware are not supported.

The DOS 3.3 CTTY Utility is not supported.

The AUX device name does not appear in the device header for the ASYNC device driver. AUX does not correspond to COM1.

We strongly recommend that applications using a serial printer from the DOS mode spool print data to the print spooler through the appropriate LPT handles.

The only support to COMn in the DOS mode is through the file system.

COM support in the DOS mode is incompatible with COM support in DOS. COM support in DOS 3.3 is half duplex through INT 14H. COM support in OS/2 has the following attributes:

- Full duplex
- Interrupt driven
- Sophisticated time-out processing
- Many different control modes of the modem control signals
- Logical flow control capabilities
- No application knowledge of state of modem control signals and data flow on a character basis.

The device driver makes no attempt to restrict or mold the function of file system requests because they may have come from DOS mode. To achieve the full capabilities of the file system access to COM, the application needs access to the full range of Category 1 IOCtls. The design and externals of the ASYNC device driver are based on the requirements of new OS/2 mode applications that use the RS232-C port of the system.

If the DOS mode application is concurrently sharing a COM port with OS/2 mode (not recommended), then care must be taken not to switch away from DOS mode while the DOS mode application has requests outstanding in the file system. Switching away from DOS mode, while file system requests are outstanding, could cause the device driver not to process certain file system requests that are outstanding (for a given port).

Refer to the SETCOM40 command in the IBM Operating System/2™ *Command Reference*.

## Control Panel Condiderations

When setting up a serial printer in the Control Panel, the user can choose between two handshaking protocols. If the user selects NONE, the serial printer card switches must be set for XON/XOFF handshaking. If the user chooses HARDWARE, the serial printer card switches must be set for DTR Pacing.

## Performance

The achievable performance is very sensitive to the environment. The type and amount of other system activity will determine the achievable performance. On the IBM PS/2 the number of COM ports or other devices on the same interrupt level will significantly affect the achievable performance level.

Trying to receive data at too high a baud rate could cause hardware overrun errors or receive queue overrun errors. Receive queue overrun errors are easily solved by adjusting the communications protocol to the size of the device driver receive queue.

Trying to transmit data at too high a baud rate also could cause the performance of OS/2 to be reduced severely.

The baud rate can be set with the MODE command or with an IOCtl. The baud rate should not be set to values which may cause receive overruns or adverse OS/2 system performance effects.

## Performance in Character (NS-16450) Mode

In Character Mode, the maximum aggregate ASYNC data transfer rate which a particular system can support (without suffering Receive Hardware Overrun errors) is based on the number of ASYNC device interrupts (Transmit and Receive) which can be serviced by the CPU in a given amount of time. Based on performance data, one way of describing this maximum data transfer rate is as follows:

- 10 MHz PS/2 sustem (Model 60) with ESDI disk drive
- System load HEAVY (disk, printer, and mouse devices active)
- ONE COM port active (COM2 - interrupt level 3)
- FULL-DUPLEX communications (simultaneous Transmit and Receive)
- MEDIUM data transfer rate (4800 bits per second)
- STANDARD ASYNC data format (10 bits per character).

Under these circumstances, the single COM port is generating 960 interrupts-per-second. This allows 1042 microseconds (on average) for the CPU to service each interrupt. The typical interrupt service time for ASYNC runs from 400 to 500 microseconds, depending on whether data is being moved, and whether the CPU has to switch from Protect to Real (or vice-versa) to move the data into the application's Read/Write buffers.

Due to the other system I/O devices being active, however, it is not at all unlikely that the CPU might occasionally not get to the serial I/O port controller to service an interrupt in time. If this happens on a receive interrupt, the data byte that just arrived in the ASYNC controller is lost. This is not an acceptable situation for high performance data communications software packages, which must re-send data packets that are received in error.

Another aspect of the performance of ASYNC is the load that an interrupting high-speed serial communications device places on the user's system. If the user has a graphics printer (or plotter) with a serial interface, and wants to print lots of graphs quickly, the user is likely to have that printer set to the highest possible baud rate, say 19200 bps. That COM port will generate 1920 interrupts-per-second (if the CPU can keep the ASYNC device driver's transmit buffer full). This very high interrupt rate will consume most of the available CPU cycles, and little other useful work will get done while the COM port is sending data to the printer.

Note that if the printer/plotter is sending data back to the host, the situation can become Full-Duplex, and data sent by the printer will occasionally be lost by the PS/2 host. This error situation is called Receive Hardware Overrun. (It could also occasionally result in the alternate error Receive Queue Overrun).

The ASYNC device driver implementation was intentionally limited to allow configurations of up to 3 COM ports, and baud rates up to 19.2K bps. But it can only support ONE PORT at 4800 BAUD whenever the system has any other I/O devices active. Higher baud rates and/or greater system loading will lead to data loss and/or low ASYNC communications throughput. Also, any serial device that is active sending and/or receiving data at high baud rates (over 4800 bps) will cause poor system performance due to its consumption of a high percentage of CPU cycles.

## Enabling Extended Hardware Buffering

In most transmit-only situations (for example, serial printers), there is always a requirement for flow control (using Output Handshaking or Automatic Transmit Flow Control - XON/XOFF). If the attached hardware can receive a significant number of characters after the modem control (pacing) signal is changed, then setting Extended Hardware Buffering ENABLED (Set Device Control Block Information - Category 1, Function 53H) may be an acceptable way to improve significantly the transmit throughput, as well as improve the operating system performance. This allows the extended hardware buffering to yield maximum serial I/O performance, while still providing the required Output Handshaking or Automatic Transmit Flow Control protocols required by the attached serial device. Testing with Extended Hardware Buffering ENABLED must be performed at the attached device when the ASYNC device driver is placed in this mode.

In many Receive and/or Transmit (half or full duplex) scenarios, disabling Input Sensitivity Using DSR will have no negative effects. Many communications devices have switches which cause DSR to be constantly **on**. Disabling Input Sensitivity Using DSR will significantly improve the ability of the serial port hardware to handle receive data without receive hardware overrun errors. Another possible approach is to set Extended Hardware Buffering ENABLED using the Set Device Control Block Information IOCtl function (or the OS/2 MODE command).

In some other Transmit and/or Receive scenarios, disabling Output Handshaking Using CTS and DSR after a communications link has been established will have no adverse effects, under normal operating conditions. Again, the achievable performance benefits could be significant. Another possible approach is to set Extended Hardware Buffering ENABLED using the Set Device Device Control Block Information IOCtl function.

The potential negative effects of disabling a default control mode of the device driver should always be thoroughly understood. The potential performance benefits, however, may far outweigh the added complexity of usage and any other potential problems. Such problems may usually be resolved either by reverting back to the Automatic Protocol Override mode, or by setting Extended Hardware Buffering DISABLED using the Set Device Control Block Information IOCtl function.

In weighing the alternatives, the per-character processing requirements must be addressed when deciding whether to override one of the device driver's default protocols. Possible data integrity problems, such as receive overrun errors, loss of data at the beginning or end of a communications session, or receiving invalid data on a noisy communications connection, can occur. Such problems must be considered before the potential performance benefits associated with extended hardware buffering can be considered.

For a port (or ports) operating at a given data transfer rate, the system will have less than 20% as much interrupt-driven device driver overhead when running with Extended Hardware Buffering ENABLED (both transmit and receive FIFO buffering active), as compared with running those ports on devices where Extended Hardware Buffering is DISABLED (due, for example, to lack of Extended Hardware Buffering capability in the serial device supporting that COM port).

Also to be considered (and equally important) are the operational characteristics of the device driver. By setting Extended Hardware Buffering ENABLED, the device driver can transmit with the full 16-character FIFO buffer active (essentially transmitting 16 characters at a time), and the Receive Data Available interrupts can provide 4, 8, or 14 characters each to the device driver's receive queue. This will be true no matter what device driver protocols are enabled (such as Output Handshaking, Input Sensitivity, or Automatic Transmit Flow Control).

Examples of scenarios where setting the FIFO Enabled mode of the device driver may be acceptable are:

- If the user does not care if too many characters are transmitted after a modem connection is broken.
- If the printer or plotter that the system is connected to will not lose data when it tells the system to stop transmitting with a modem control signal and the system continues to transmit a significant number (up to 16) of characters.
- If the system is connected to a modem or another system that is normally set up to always keep DSR ON.
- If the communications protocol with the remote device does not require the system to respond on a character-by-character basis to input data (such as when the remote device sends data in blocks and provides error-retry capability on a block basis rather than a per-character basis).

# Pointer Draw (Screen) Device Driver

The Mouse Pointer Draw (Screen) device driver draws the system default and user-supplied mouse pointer images on the screen.

In the OS/2 mode, the pointer draw device driver (POINTDD.SYS) provides two levels of display mode support:

- Full draw support
- Disabled draw support.

## Full Draw Support

Full draw support includes the full pointer drawing capabilities of a pointer device driver. The following display modes are supported at this level:

- Text only

  Modes 0, 1, 2, 3 and their + and * variations.
  Mode 7 and its + variation.

These modes are always accepted by the CheckModeProtect function.

If a mode other than one of the above is set using VioSetMode and pointer drawing is not disabled, the pointer device driver will report an error to the mouse device driver causing the mouse device driver to shut down. When the Mouse device driver is shut down, the API is active but no interrupt data can be returned. In this case, a status flag is set and is available using the MouGetDevStatus function.

## Disabled Draw Support

Disabled Draw support is available when the pointer draw functions have been disabled by the MouSetDevStatus function. When this occurs, an extended set of display modes is supported by the CheckModeProtect function. These display modes are:

- Text modes

  modes 0, 1, 2, 3 and their + and * variations.
  mode 7 and its + variation.

- Graphics modes

  modes 4, 5, 6, 7, D, E, F, 10, 11, 12, and 13.

- The IBM Personal System/2™ Display Adapter advanced function modes.

In this disabled state, no pointer drawing is performed by the pointer draw device driver. Instead, the application is expected to perform the pointer draw functions. Subsequent MouSetDevStatus function calls to re-enable the pointer drawing will continue to be bypassed until VioSetMode resets the screen mode to one of the supported modes.

## DOS Mode Support

In the DOS mode, the pointer draw device driver (POINTDD.SYS) provides full pointer draw functions for the following modes:

- Text modes

  modes 0, 1, 2, 3, and their + and * variations.
  mode 7 and its + variation.

- Graphics modes (hex)

  modes 4, 5, 6. D, E, F, 10, 11, 12, and 13.

All other modes will cause the mouse device driver to shut down.

## Mouse Pointer Draw Implementation

Communications between the Mouse Device Driver and a session's Pointer Draw Device Driver is conducted by way of a FAR call from the Mouse Device Driver to the Pointer Draw Device Driver's IDC Entry Point routine. This communication occurs for both the DOS and OS/2 modes of mouse support operations.

The setup required for the mouse device driver prior to issuing the call to the pointer draw routine is as follows:

- Set DS:SI to point to the session data control block described below.
- Set the screen_func field to indicated the desired function code.
- Issue a FAR call to the pointer draw routine. The address to be called must be stored in the screen_entp field of the mouse session data control block.

When the mouse device driver calls the screen pointer draw routine, the draw routine must issue a CLI to disable interrupts. The mouse device driver will have disabled the IOCtls and the mouse device interrupts. These will be enabled by the mouse device driver after the pointer draw routine returns to it.

All addresses to data obtained through a AllocPhys DevHlp call are stored in the control blocks in 32 bit physical address form. The pointer draw routine must convert these addresses to virtual format (selector:offset for OS/2 mode and segment:offset for DOS mode) by calling the PhysToVirt DevHlp. The resulting selector is temporary and will be valid for no more than 400 microseconds. If an interrupt occurs during the 400 microsecond span, the temporary selector becomes invalid.

Consequently, the screen pointer draw routine must:

- Disable interrupts using the CLI instruction
- Complete all operation within the 400 microsecond time limit
- Reenable interrupts.

It is possible for the screen pointer draw routine to do repetitively the following in order to bypass the 400 microsecond time limit:

- Enable interrupts
- Disable interrupts
- Call DevHlp_PhysToVirt
- Execute draw functions.

However, interrupt time operations should be as limited in scope and duration as possible to reduce the impact on the remainder of the system. Therefore, all effort should be directed to completing the pointer draw operations as quickly as possible and without exceeding a single 400 microsecond interval.

The functions to be supported by the screen pointer draw routine are:

- screen_func = 0 = DrawPointer (Bimodal)
- screen_func = 1 = RemovePointer (Bimodal)
- screen_func = 2 = FreePointerMemory (Bimodal)
- screen_func = 3 = CheckModeProtect (OS/2 mode only)
- screen_func = 4 = CheckModeReal (DOS mode only)
- screen_func = 5 = GetPointerMemory (Bimodal).

Table 6-1 outlines the execution modes from which the commands may be invoked.

Table 6-1 (Page 1 of 2). Call Modes

| Function Type | Call Modes |
|---|---|
| DrawPointer | Interrupt, User, Kernel |
| RemovePointer | Interrupt, User, Kernel |
| FreePointerMemory | User, Kernel |

| Table 6-1 (Page 2 of 2). Call Modes | |
|---|---|
| **Function Type** | **Call Modes** |
| CheckModeProtect | Kernel |
| CheckModeReal | User |
| GetPointerMemory | User, Kernel |

Functions that are supported by the Pointer Draw IDC Interface follow:

**DrawPointer:** This function draws the current mouse pointer according to the pointer image definitions and positioning data contained within the Mouse Session Control Block. DrawPointer is also responsible for saving the current screen data that the mouse pointer image will overwrite.

**Calling Sequence**

1. DS:SI points to the MSCB.
2. Set the pointer image row and column position fields (Ptr_Row_Pos and Ptr_Col_Pos) in the MSCB to the coordinates where the pointer image is to be drawn.
3. Set the MSCB screen function field to 0.
4. Execute a far call to the Pointer Draw device driver entry point address.

**RemovePointer:** This function removes the current pointer image from the display. Before issuing the call, an advisory parameter (CX) should be set as follows:

**CX = 0** The pointer image is removed before returning to the Mouse device driver.

**CX = 1** The pointer image is not removed until the next DrawPointer function call. The Pointer Draw device driver saves the current pointer image screen positions for the upcoming image removal. This increases performance and reduces pointer image flicker. This can be used with display adapters that have built in pointer image drawing hardware.

**Calling Sequence**

1. DS:SI points to the MSCB.
2. Set the MSCB function field to 1.
3. Set CX to indicate the type of pointer image removal desired. (0 = unconditional removal, 1 = postponed removal)
4. Execute a far call to the Pointer Draw device driver entry point address.

The maintenance of pointer image save and restore buffers, handled by the Pointer Draw device driver, is implicit in both DrawPointer and RemovePointer functions. The Pointer Draw device driver must always save the current display screen contents before drawing the pointer image. The Pointer Draw device driver removes the pointer image by restoring the save buffer contents.

**FreePointerMemory:** This function tells the Pointer Draw device driver to free both the current pointer image buffer and it's associated screen restore buffer. This request can be made even if there are no allocated buffers to be freed. This function always returns with AX=0 and restores a default pointer image appropriate for the display mode.

**Calling Sequence**

1. DS:SI points to the MSCB.
2. Set the MSCB screen function field to 2.
3. Execute a far call to the Pointer Draw device driver entry point address.

**CheckModeProtect:** This function verifies display mode data for Protected Mode sessions. The Mouse device driver uses this function to determine what level of support is available for a particular display mode. The levels of support are as follows:

| | |
|---|---|
| **Unsupported** | The Pointer Draw device driver does not recognize the display mode/config data. |
| **Disabled Draw Support** | The Pointer Draw device driver recognizes the display mode/config data, but can't support drawing operations. For a non-error return, the MSCB device status word (bit 1) must be set to indicate that Pointer Draw operations are disabled. This function returns an error condition of AX = 1 if drawing operations have not been disabled. This is a partially supported mode. |
| **Full Draw Support** | The Pointer Draw device driver can provide both tracking data and pointer image drawing operations for the input display mode/config data. |

If the display mode/config data is supported or partially supported, the Pointer Draw device driver calculates the row and column cell sizes and stores them in the MSCB before returning to the Mouse device driver.

The Mouse Device Driver copies the mode data into the MSCB. The Pointer Draw Device Driver fills in the row and column cell sizes if the mode is supported or partially supported. The Pointer Draw device driver should not alter any of the mode data structure fields in the MSCB or any fields that are pointed to by the extended mode data address field.

The mode data is stored in non-contiguous memory by the Mouse device driver. The Mouse device driver always provides a contiguous VIOSetMode data record on every CheckModeProtect call. The Pointer Draw device driver can access any extended mode data by using the extended mode data pointer fields in the MSCB. Any imbedded addresses in the VIOSetMode data structure must be checked by the Pointer Draw device driver before they are used. The Mouse device driver does not save data that imbedded addresses point to. The fields are only valid during a CheckModeProtect call. The Pointer Draw device driver must save this data if it is needed. The config data pointer in the MSCB is maintained by the Mouse device driver so that it always points to the current display's configuration data.

**Calling Sequence**

1. DS:SI points to the MSCB.
2. ES:DI points to the VIOSetMode data record.
3. Set the MSCB screen function field to 3.
4. Set the MSCB config data pointer to the VIOGetConfig data record.
5. Execute a far call to the Pointer Draw device driver entry point address.

**CheckModeReal:** This function verifies display mode data for the real mode or DOS 3.X compatibility session. The Mouse device driver uses this function to determine if the current display mode is supported by the Pointer Draw device driver. The two levels of support for real mode are:

| | |
|---|---|
| **Unsupported** | The Pointer Draw device driver does not support this display mode. |
| **Supported** | The Pointer Draw device driver can provide both tracking data and full pointer image drawing support for this display mode. |

The Pointer Draw device driver calculates the row and column cell sizes and stores them in the MSCB. The Pointer Draw device driver also calculates the virtual screen row and column resolution sizes and stores them in the input set mode data structure.

CheckModeReal fills in the MSCB with the current mode data. This is different from CheckModeProtect. The Mouse device driver pulls the request's display mode data from the BIOS Data Areas after an INT 10H SetMode or SetFont function request. The Pointer Draw device driver fills in the base mode data fields of the MSCB. The MSCB contains a display config data pointer for real mode as it does for protected mode sessions. However, the Pointer Draw device driver is pro-

hibited from using any of the nested data pointers located within the Config Data record for this call. The config data pointer in the MSCB is saved as a physical address to the power-up display config record. The Pointer Draw device driver converts the physical address to a virtual address if access is required to the display config data record.

**Calling Sequence**

1. DS:SI points to the MSCB.
2. ES:DI points to the DispMode data record.
3. Set the MSCB config data pointer to the VIOGetConfig data record.
4. Execute a far call to the Pointer Draw device driver entry point address.

**GetPointerMemory:** This function, used by the Mouse device driver, tells the Pointer Draw device driver to allocate memory for the new pointer image and screen save/restore buffers.

**Calling Sequence**

1. DS:SI points to the MSCB.
2. ES:DI points to the PtrRec data record.
3. Set the MSCB screen function field to 5.
4. Execute a far call to the Pointer Draw device driver entry point address.

**Note:** See the Pointer Draw IDC interface data structures section for specific parameter structure definition.

FreePointerMemory and GetPointerMemory are used together. For example, if a new pointer image is to be set; first a FreePointerMemory request would be issued to release the current pointer buffers. A GetPointerMemory request would then be issued to allocate buffers for the new pointer. If a new pointer image is being defined it is important to 1st remove the current pointer image on the screen, and then define the new pointer image buffer.

## Pointer Draw IDC Interface Data Structures

The following section describes the data structures that are passed as parameters in the Pointer Draw Device Driver IDC Interface.

**DispMode Structure:** This structure is used in the CheckModeReal IDC request function. The first three fields are copied from the real mode (compatibility box) BIOS data area. This structure data indicates the mode to which the compatibility box just set, either by a INT 10H, AH=0 (setmode) call or INT 10H, with AH=11H and AL=1xH (character generator). The Pointer Draw device driver returns the new Display Mode's virtual coordinate maximum grid values by way of the virt_rows and virt_cols fields.

```
;
;     IDC Data Structure used for CheckModeReal Function call
;
DispMode   STRUC                  ; For Real Mode Support
           R_Mode       DB     ; Standard Disp Mode, CGA
           Ex_Rows      DB     ; Extended Mode Display Rows
           Ex_Points    DW     ; Extended Mode Display Cols
           Virt_Rows    DW     ; Mode's Virtual Disp Row Resolution
           Virt_Cols    DW     ; Mode's Virtual Disp Col Resolution
DispMode   Ends
```

For OS/2 mode see VioSetMode for the structure provided on a CheckModeProtect.

**PtrRec Structure:** This structure is used in the GetPointerMemory request function. It contains two address fields which point to data areas used to define the new pointer image.

```
;
;     IDC Data Structure used for GetPointerMemory Function call
;
PtrRec     STRUC
           addr1          DD     ; Address to ptr definition record
           addr2          DD     ; Address to ptr image buffer
PtrRec     ENDS
```

**Addr1 field of PtrRec:** The Addr1 field is a pointer to the Pointer Image Definition Record (PtrDefTed).

```
;
;     Pointer Image Definition Record Structure
;
PtrDefRec  STRUC
           Buf_Len     DW     ; Ptr Shape Buffer Length (Bytes)
           Width       DW     ; Pointer shape width dimension
           Height      DW     ; Pointer shape height dimension
           Col_Hot     DW     ; Ptr column hot spot pixel index
           Row_Hot     DW     ; Ptr row hot spot pixel index
PtrDefRec  ENDS
```

For text and mono mode pointers, the OS/2 Pointer Draw Device Driver only supports pointer images which are defined with the following:

- Buf_Len = 4 (see Calculating Buf_Len below)
- Width = 1 (must be a single character cell width)
- Height = 1 (must be a single character cell height)
- Col_Hot = 1 (single character cell limitation)
- Row_Hot = 1 (single character cell limitation).

**Note:** Hot Spot values are only used in graphics images to determine the pixel position within the pointer image. This is considered the focal point for coordinate tracking purposes.

**Calculating Buf_Len:** The equation used to calculate the total number of bytes (Buf_Len) needed for a given text mode pointer is:

```
Buf_Len = (height in chars) * (width in chars) * (number of image masks (AND and XOR)) *
          (number of data bytes (character and attribute)).
```

The OS/2 Pointer Draw device driver limits it's text mode pointer images to a single character cell in size. The following is an example of a Buf_Len text mode calculation:

```
Buf_Len =        1 (character width)
               * 1 (character height)
               * 2 (number of image masks = (1 AND mask + 1 XOR
                   mask))
               * 2 (number of data bytes = (1 character + 1
                   attribute))
                 ---
                 4 Bytes
```

For graphics mode pointers, the OS/2 Pointer Draw device driver only provides drawing operations in the compatibility box environment. The field meanings are in pels for graphics modes instead of characters, which are used in text modes.

The OS/2 Pointer Draw device driver has some restrictions for defining graphics pointer images in order to maximize pointer image drawing performance. These restrictions are as follows:

- Graphics modes utilizing the 320x200 resolution require that pointer images be defined with 4 pixels per byte.
- Graphics modes utilizing the 640x200 resolution require that pointer images be defined with 8 pixels per byte.
- Graphics images must always have the width in pels specified in multiples of 8 bits.
- Graphics pointer images are only supported on a single display plane.

Graphics pointer images must be defined in byte-width multiples. Non-byte width definitions may be accepted by the OS/2 Pointer Draw device driver but may also result in peculiar pointer images appearing on the display screen.

The Pointer Draw equation for calculating the number of bytes (Buf_Len) needed for a graphics pointer image is:

```
Buf_Len = (height in pels) * (width in pels) * (bits per pel) *
          (number of image masks (AND and XOR)) / (bits per byte)
```

The following are examples of Buf_Len calculations for graphics mode:

For CGA display modes 4 and 5 or compatible, which have display resolutions of 320x200, a Buf_Len calculation would look like this:

```
Buf_Len =        16 (height in pels)
               *  8 (width in pels)
               *  2 (bits per pel)
               *  2 (number of image masks = (1 AND mask + 1 XOR
                    mask))
               /  8 (bits per byte)
                  ——

                 64 Bytes
```

For CGA display mode 6 or compatible, which has a display resolution of 640x200, a Buf_Len calculation would look like this:

```
Buf_Len =        16 (height in pels)
               * 16 (width in pels)
               *  1 (bits per pel)
               *  2 (number of image masks = (1 AND mask + 1 XOR
                    mask))
               /  8 (bits per byte)
                  ——

                 64 Bytes
```

The hotspot values for the graphics default pointer images are:

- For CGA display modes 4 and 5 or compatible, Row_Hot = 1 and Col_Hot = 5
- For CGA display mode 6 or compatible, Row_Hot = 1 and Col_Hot = 11.

**Addr2 field of PtrRec:** The Addr2 field is a pointer to the actual pointer image buffer. The pointer image buffer format depends on it's associated display mode and configuration that it is to be drawn on. The OS/2 Pointer Draw device driver pointer image buffer always consists of the:

- AND pointer image data
- XOR pointer image data.

XOR always follows AND. The buffer always describes only one display plane.

The OS/2 Pointer Draw device driver default text mode pointer image buffer is defined as follows:

```
DefTxtPtr     DW   0FFFFH,0770H        ; default text pointer data
```

The OS/2 Pointer Draw device driver default graphics mode pointer image buffer, for CGA modes 4, 5, and 6, is defined as follows:

```
DefGphPtr     DB   11111111B,11000011B  ; default graphics pointer data
              DB   11111111B,10000011B  ; AND mask
              DB   11111111B,00000011B
              DB   11111110B,00000011B
              DB   11111100B,00000011B
              DB   11111000B,00000011B
              DB   11110000B,00000011B
              DB   11100000B,00000011B
              DB   11000000B,00000011B
              DB   10000000B,00000011B
              DB   11110000B,00000011B
              DB   11110000B 00000011B
              DB   11100000B,00111111B
              DB   11100000B,01111111B
              DB   11100000B,01111111B

              DB   00000000B,00000000B  ; XOR mask
              DB   00000000B,00010000B
              DB   00000000B,00110000B
              DB   00000000B,01110000B
              DB   00000000B,11110000B
              DB   00000001B,11110000B
              DB   00000011B,11110000B
              DB   00000111B,11110000B
              DB   00001111B,11110000B
              DB   00011111B,11110000B
              DB   00000001B,11110000B
              DB   00000011B,00010000B
              DB   00000011B,00000000B
              DB   00000110B,00000000B
              DB   00000110B,00000000B
              DB   00000000B,00000000B
```

## Mouse Session Control Block (MSCB)

This structure is owned and used by the Mouse device driver to keep track of each session. On entry to every Pointer Draw IDC function, DS:SI contains the address to the MSCB. The following is the MSCB format:

```
;
; Mouse Screen Group Control Block (132 byte structure)
;
MSCB   STRUC
;
;   Session CB data sub-table (40 bytes)
;   Section Access Rights: Mouse = r/w, Pointer Draw = r
;
      RowScale_Fact     DW   ; Row movement scaling factor
      ColScale_Fact     DW   ; Column movement scaling factor
      Row_Remain        DW   ; Row motion pixel remainder
      Col_Remain        DW   ; Col motion pixel remainder
      D_Status          DW   ; Device status flags
                             ; Low Byte:
                             ;  0000xxxx Reserved = 0
                             ;  xxxx1xxx Unsupported display mode set
                             ;  xxxx0xxx Supported display mode active
                             ;  xxxxx1xx Event queue flush in progress
                             ;  xxxxx0xx Event queue flush not active
                             ;  xxxxxx1x Process blocked on event data
                             ;  xxxxxx0x No Process blocked on events
                             ;  xxxxxxx1 Event queue busy (read/write)
                             ;  xxxxxxx0 No Event queue activity
      E_Mask            DW   ;Event mask-selects events to be queued
                             ; High Byte:
                             ;  00000000 Reserved = 0
                             ; Low Byte:
                             ;  0xxxxxxx Reserved = 0
                             ;  x1xxxxxx Button 3 events without motion
                             ;  xx1xxxxx Button 3 events with motion
                             ;  xxx1xxxx Button 2 events without motion
                             ;  xxxx1xxx Button 2 events with motion
                             ;  xxxxx1xx Button 1 events without motion
                             ;  xxxxxx1x Button 1 events with motion
                             ;  xxxxxxx1 Motion only events
      Hdle_Cntr         DW   ; Number of open device handles
      E_Queue           DW   ; Offset to start of event queue
      Eq_Head           DW   ; Event queue head offset pointer
      Eq_Tail           DW   ; Event queue tail offset pointer
      Eq_PID            DW   ; PID blocked on event queue
      Eq_Size           DW   ; Number of elements in queue
      Chain_Size        DW   ; Number of monitors in chain
      Chain_Hdle        DW   ; Monitor chain handle
      Screen_Entp       DW   ; Pointer Draw entry point address
      Screen_Tble       DW   ; Pointer Draw work area address
      Screen_Func       DW   ; Pointer Draw IDC function code
      Screen_DOff       DW   ; Pointer Draw DS offset, = 0
      Screen_DSeg       DW   ; Pointer Draw DS (segment/selector)
;
;   Monitor Chain Output Buffer (14 Bytes)
;   Section Access Rights: Mouse and Monitor Dispatcher = r/w.
;
      MB_Len            DW   ; Monitor output buffer length (bytes)
      MFlags            DW   ; Monitor flags
      EMask             DW   ; Event's mouse event mask
      EMTime            DD   ; Event's millisecond time stamp
      Row_Pos           DW   ; Event's ptr_row motion/coord data
      Col_Pos           DW   ; Event's ptr_col motion/coord data
;
;   Base Display Mode Data (12 Bytes)
;   Section Access Rights: Mouse and Pointer Draw = r/w
;
      MLength           DW   ; Total display mode data size (bytes)
      MType             DB   ; Mono text/color text/color graphics
      Color             DB   ; Number of color bits (graphics type only)
      TCol_Res          DW   ; Column resolution (text)
      TRow_Res          DW   ; Row resolution (text)
      GCol_Res          DW   ; Column resolution (graphics)
      GRow_Res          DW   ; Row resolution (graphics)
;
```

```
;   Display Mode Cell Sizes (4 bytes)
;   Section Access Rights: Mouse and Pointer Draw = r/w
;
      Col_Cell_Size      DW   ; (Graphics col res)/(text col res)
      Row_Cell_Size      DW   ; (Graphics row res)/(text row res)
;
;   Mouse/Pointer Draw fields (38 Bytes)
;   Section Access Rights: Pointer Draw = r/w, Mouse = r and
;   Mouse = r/w for Ptr_Flags, Ptr_Row_Pos, and Ptr_Col_Pos
;
      Ptr_Flags          DW   ; Pointer image activity flags
                              ;  High Byte:
                              ;   00000000 Reserved = 0
                              ;  Low Byte:
                              ;   00000xxx Reserved = 0
                              ;   xxxxx1xx Drawing operations disabled
                              ;   xxxxx0xx Drawing operations enabled
                              ;   xxxxxx1x Vertical retrace required
                              ;   xxxxxx0x Vertical retrace not required
                              ;   xxxxxxx1 Ptr image is visible
                              ;   xxxxxxx0 Ptr image is hidden
      Ptr_Height         DW   ; Height of ptr image (resolution units)
      Ptr_Width          DW   ; Width of ptr image (resolution units)
      Ptr_Row_Pos        DW   ; Current pointer row coord position
      Ptr_Col_Pos        DW   ; Current pointer col coord position
      Ptr_Rowhot         DW   ; Row index ptr shape reference pixel
      Ptr_Colhot         DW   ; Col index ptr shape reference pixel
      Ptr_Bufflow        DW   ; Physical address to ptr image buffer
      Ptr_Buffhigh       DW   ; Physical address High word
      Ptr_Buf_Len        DW   ; Pointer image total buffers len (bytes)
      Ptr_Imagelen       DW   ; Pointer image buffer length (bytes)
      Ptr_Imageoff       DW   ; Pointer image offset (bytes)
      Ptr_Linelen        DW   ; Pointer image line length
      Ptr_Skiplen        DW   ; Pointer image skip length
      Tot_Linelen        DW   ; Pointer total line length
      Ptr_Savstart       DW   ; Pointer save start pointer
      Ptr_Savend         DW   ; Pointer save end pointer
      Ptr_Savstatodd     DW   ; Pointer save start odd pointer
      Ptr_Savendodd      DW   ; Pointer save end odd pointer
;
;   Collision Area Fields (10 Bytes)
;   Section Access Rights: Mouse = r/w, Pointer Draw = r
;
      Area_Flags         DW   ; Collision area definition flags
                              ;  High Byte:
                              ;   00000000 Reserved = 0
                              ;  Low Byte:
                              ;   0000000x Reserved = 0
                              ;   xxxxxxx1 Collision area defined
                              ;   xxxxxxx0 No collision are defined
      Area_Top           DW   ; Starting row coord position
      Area_Left          DW   ; Starting col coord position
      Area_Bot           DW   ; Ending row coord position
      Area_Right         DW   ; Ending col coord position
;
;   Cell Remainder (4 Bytes)
;   Section Access Rights: Mouse = r/w
;
      Col_Cell_Remain    DW   ; Column cell remainder
      Row_Cell_Remain    DW   ; Row cell remainder
;
;   Extended Display Mode data pointer (4 Bytes)
;   Section Access Rights: Mouse = r/w, Pointer Draw = r
;
      ExtModePtr         DD   ; Pointer to any extended mode data
;
;   Display Configuration data (6 Bytes)
;   Section Access Fights:  Mouse = r/w, Pointer Draw = r

;
      Cur_Config         DW   ; Current display configuration number
      CfgOffset          DW   ; Offset to display configuration data
      CfgSelector        DW   ; Selector to display configuration data
MSCB  ENDS
```

**Mouse Screen Resolutions:** The screen resolution is determined by system default or by the application issuing an explicit:

- VioSetMode call in OS/2 mode
- INT 10H, AH = 0 in the DOS mode. eul.

When in DOS mode the virtual display resolution is used. The virtual display resolution depends on the display mode selected.Table 6-2 lists the display modes supported in DOS mode only.

Table 6-2. Virtual Display Resolution

| Mode | Type | Text Resolution | Graphics Resolution | Virtual (X,Y) Coordinates | Cell Size |
|---|---|---|---|---|---|
| 0 | BW Text | 40 x 25 | 320 x 200 | 640 x 200 | 8 x 8 |
| 0+ | BW Text | 40 x 25 | 320 x 350 | 640 x 200 | 8 x 14 |
| 0* | BW Text | 40 x 25 | 360 x 400 | 640 x 200 | 9 x 16 |
| 1 | CO Text | 40 x 25 | 320 x 200 | 640 x 200 | 8 x 8 |
| 1+ | CO Text | 40 x 25 | 320 x 350 | 640 x 200 | 8 x 14 |
| 1* | CO Text | 40 x 25 | 360 x 400 | 640 x 200 | 9 x 16 |
| 2 | BW Text | 80 x 25 | 640 x 200 | 640 x 200 | 8 x 8 |
| 2+ | BW Text | 80 x 25 | 640 x 350 | 640 x 200 | 8 x 14 |
| 2* | BW Text | 80 x 25 | 720 x 400 | 640 x 200 | 9 x 16 |
| 3 | CO Text | 80 x 25 | 640 x 200 | 640 x 200 | 8 x 8 |
| 3+ | CO Text | 80 x 25 | 640 x 350 | 640 x 200 | 8 x 14 |
| 3* | CO Text | 80 x 25 | 720 x 400 | 640 x 200 | 9 x 16 |
| 4 | Graphics | - | 320 x 200 | 640 x 200 | 2 x 1 |
| 5 | Graphics | - | 320 x 200 | 640 x 200 | 2 x 1 |
| 6 | Graphics | - | 640 x 200 | 640 x 200 | 1 x 1 |
| 7 | Mono | 80 x 25 | 720 x 350 | 640 x 200 | 9 x 14 |
| 7+ | Mono | 80 x 25 | 720 x 400 | 640 x 200 | 9 x 16 |
| D | Graphics | - | 320 x 200 | 640 x 200 | 2 x 1 |
| E | Graphics | - | 640 x 200 | 640 x 200 | 1 x 1 |
| F | Graphics | - | 640 x 350 | 640 x 350 | 1 x 1 |
| 10 | Graphics | - | 640 x 350 | 640 x 350 | 1 x 1 |
| 11 | CO Graphics | - | 640 x 480 | 640 x 480 | 1 x 1 |
| 12 | CO Graphics | - | 640 x 480 | 640 x 480 | 1 x 1 |
| 13 | CO Graphics | - | 320 x 200 | 320 x 200 | 1 x 1 |

To calculate a text mode cell size for both row and column, divide the Graphics Resolution by the Text Resolution. The Virtual Coordinates listed above are the default values for each display mode. When a non-default font is loaded for a display mode, the cell sizes change accordingly. However, to update the Virtual Display coordinates, an INT 33H Function 8 call must be made by the application to regain full screen pointer motion.

# Mouse Device Driver

## Mouse Device Overview

OS/2 provides support for a pointing device (mouse) for DOS mode and OS/2 mode applications through four components:

- A mouse device driver
- A mouse router
- A base mouse subsystem
- A pointer draw device driver.

**Mouse Devices Supported:** Mouse device drivers are supported in both DOS mode and OS/2 mode for the following devices:

- Microsoft®[1] Bus (parallel) Mouse for IBM Personal Computers (part number 037-099, 100ppi)
- Microsoft® Bus (parallel) Mouse for IBM Personal Computers (part number 037-199, 200ppi)
- Microsoft® Mouse (serial) for IBM Personal Computers (part number 039-099, 100ppi)
- Microsoft® Mouse (serial) for IBM Personal Computers (part number 039-199, 200ppi)
- PC Mouse™[2] (serial) by Mouse Systems (part number 900120-214, 100ppi)
- Visi On™[3] Mouse (serial) (part number 69910-1011, 100ppi)
- Microsoft® Mouse (InPort) for IBM Personal Computers (part number 037-299, 200ppi)
- PS/2 Mouse (In-Processor) for IBM Personal System/2™ Computers (part number 6450350, 200ppi)
- Microsoft® Mouse configurable as a PS/2, Serial, or InPort Mouse (part number 056-x99).

While support is provided for both DOS mode and OS/2 mode applications, DOS mode applications access the mouse device differently from OS/2 mode application access. DOS mode applications use the interrupt 33H (INT 33H) interface described later in this chapter. OS/2 mode applications use the OS/2 mode MOU*xxx* API also described in this chapter and in the *OS/2 Version 1.2 Programming Reference* and in the *OS/2 Version 1.2 Programming Guide*. OS/2 mode applications may not use the INT 33H API and may DOS mode applications may not use the Mou*Xxx* mouse interface.

System Install ensures that mouse device driver initialization takes place prior to ASYNC device driver initialization. This allows the ASYNC device driver to determine that it is not responsible for servicing that port. This ensures that mouse device drivers are not preempted from the COMx ports by the ASYNC device drivers.

**Note:** When manually changing CONFIG.SYS, the user must place the mouse DEVICE= statements before ASYNC DEVICE= statements.

**Mouse Device Driver Packaging:** OS/2 Version 1.2 mouse device driver support consists of two parts:

- A device-independent mouse device driver responsible for all hardware independent mouse operations.
- A device-dependent mouse device driver responsible for all hardware dependent operations.

The device-independent mouse device driver is a single generic device driver, loadable across all OS/2-supported hardware families. The generic device driver filename is MOUSE.SYS and the device name is MOUSE$. The generic mouse device driver provides the system level interface for mouse support.

---

1 Microsoft is a registered trademark of Microsoft Corporation

2 PC Mouse is a trademark of Metagraphics/Mouse Systems

3 Visi On is a trademark of Visi On Corporation

**Note:** The device-independent mouse device driver is not a replaceable OS/2 device driver.

Device-dependent mouse device drivers follow the standard OS/2 device driver naming convention of xxxxxx0#.SYS for device driver filenames. The currently supported Mouse devices and their device-dependent device names and device driver filenames follow:

| Mouse Device/Part Number | Device Name | DD Filename | H/W Family |
|---|---|---|---|
| PC Mouse Systems<br>PN 900170-214 | PCMOU$ | PCMOU01.SYS | PC-AT<br>PS/2 MOD 30-286 |
| | | PCMOU02.SYS | PS/2 |
| Visi-On<br>PN 69910-1011 | VISION$ | VISION01.SYS | PC-AT<br>PS/2 MOD 30-286 |
| | | VISION02.SYS | PS/2 |
| Microsoft Serial/039-099<br>PN 039-099,039-199 | MSSER$ | MSSER01.SYS | PC-AT<br>PS/2 MOD 30-286 |
| | | MSSER02.SYS | PS/2 |
| Microsoft Bus<br>PN 037-099,037-199 | MSBUS$ | MSBUS01.SYS | PC-AT<br>PS/2 MOD 30-286 |
| Microsoft Inport<br>PN 037-299 | MSINP$ | MSINP01.SYS | PC-AT<br>PS/2 MOD 30-286 |
| IBM In-Processor<br>PS/2 Mouse | IBMMOU$ | IBMMOU01.SYS<br>IBMMOU02.SYS | PS/2 MOD 30-286<br>PS/2 |
| Microsoft Mouse<br>PN 056-x99<br>Configured as InPort | MSINP$ | MSINP01.SYS<br>$rbl. | PC-AT<br>PS/2 MOD 30-286 |
| Microsoft Mouse<br>PN 056-x99<br>Configured as Serial | MSSER$ | MSSER01.SYS<br>MSSER02.SYS * | PC-AT<br>PS/2 MOD 30-286<br>PS/2 |
| Microsoft Mouse<br>PN 056-x99<br>Configured as PS/2 | MSPS2$ | MSPS201.SYS<br>MSPS202.SYS | PC-AT<br>PS/2 MOD 30-286<br>PS/2 |

**Note:** OS/2 mouse device drivers from previous versions of OS/2 are not compatible with OS/2 Version 1.2. OS/2 mouse device drivers from previous versions of OS/2 may install successfully; however, unexpected and unrecoverable results occur.

**Mouse Installation:** Mouse support is installed at IPL (start-up) time. Mouse support may be tailored to the user's needs. This is accomplished by using the CONFIG.SYS file to define system mouse requirements.

To load OS/2 mouse support, there must be a minimum of three DEVICE= statements in the CONFIG.SYS file:

- An entry for the device-dependent mouse device driver

  **Format** DEVICE=DDFName [SERIAL=] [MODEL=] [DD unique keywords"

  **Example** DEVICE=MSSER01.SYS SERIAL=COM2 MODEL=199

- An entry for the device-independent mouse device driver

  **Format** DEVICE=MOUSE.SYS [TYPE=] [MODE=] [QSIZE=]

  **Example** DEVICE=MOUSE.SYS TYPE=MSSER$ MODE=P QSIZE=15

- An entry for at least one Pointer Draw Device Driver

* Must use the MODEL=199 parameter for proper support.

**Format** DEVICE=DDFName [DD unique keywords]

**Example** DEVICE=POINTDD.SYS

**Note:** The DEVICE= statement for the device-dependent mouse device driver must precede the DEVICE= statement for the device-independent mouse device driver. These DEVICE= statements must precede any ASYNC DEVICE= statements.

The following table describes the CONFIG.SYS DEVICE= keyword parameters for the device-dependent mouse device drivers.

| Keyword | Function | Parameter Range | Default |
|---|---|---|---|
| SERIAL= | Communications Port for serial mice | COM1-COM8 (PS/2)<br>COM1-COM2 (PC-AT)<br>Non-serial devices | COM1<br>COM1<br>None |
| MODEL= | Device Dependent model type selection | 099 and 199 valid only for MSSER$ and MSBUS$ | 99 |

**Note:** Keywords may be in upper, lower, or mixed case, but must be in English.

The following table describes the CONFIG.SYS DEVICE= keyword parameters for the device independent mouse device drivers:

| Keyword | Function | Parameter Range | Default |
|---|---|---|---|
| MODE= | Processor operating modes supported | P (OS/2 mode only)<br>R (DOS mode only)<br>B (Both modes) | B |
| QSIZE= | OS/2 mode support event queue size in elements | 1 - 100 elements | 10 |
| TYPE= | indicates which mouse to support - Device name (character string with $ terminator) | None (This parameter is required.) | |

**Note:** Keywords may be in upper, lower, or mixed case, but must be in English words.

The TYPE=keyword is used to connect the device-independent and device-dependent device driver. Together, these drivers form mouse support.

## Mouse Inter Device Driver (IDC) Interfaces

**Overview:** The OS/2 Mouse Device Driver support is split into two (2) pieces: device-dependent and device-independent. The device-independent piece is packaged in the same manner as the previous release of OS/2. The device-dependent pieces are packaged as separate device drivers.

The device-independent support is a single generic device driver that is loadable across all OS/2 supported hardware families. This generic device driver filename is MOUSE.SYS.

The device-independent mouse device driver provides the system level interface for mouse support for:

- the OS/2 mode IOCtl interface
- the DOS mode INT 33H interface.

In addition, the device-independent mouse device driver contains support for IDC (inter-device-driver communication) interfaces with:

- the OS/2 pointer draw device driver
- the device-dependent mouse device drivers.

The device-dependent mouse device drivers contain support for:

- Device specific initialization

- Data reformatting responsibilities
- The generic mouse IDC interface
- Strategy handling for a minimal set of OS/2 Device Driver commands (INIT and DEINSTALL).

**Operational Description:** All mouse support is loaded through CONFIG.SYS processing. The device-dependent device driver must be loaded prior to the device-independent mouse device driver.

The device-independent mouse device driver, MOUSE.SYS, uses CONFIG.SYS TYPE parameter to connect to the device-dependent device driver. The MODE and QSIZE keywords are valid input parameters for MOUSE.SYS processing. However, the SERIAL and MODEL keywords are valid only for the device-dependent device drivers, which support those options, and must appear on that device driver's CONFIG.SYS line entry.

When the device-dependent mouse device driver initializes, it must:

1. Validate the machine base that it is loaded on (PC-AT, PS/2, PS/2 MOD 30-286)
2. Validate the device presence (card, ports, etc.)
3. Claim the device IRQ# as shared/non-shared based on machine type.
4. Suppress the device until IDC Initialization completes
5. Process any optional CONFIG.SYS parameters
6. Handle special case operations.

When the device-independent mouse device driver initializes, it must:

1. Process CONFIG.SYS parameters
2. Issue an AttachDD device helper call on the required name specified in the TYPE parameter

   This establishes communications between the generic device driver and the device-dependent device driver.

As the device-dependent device driver processes device interrupts, it places the data into a shared data buffer residing in the device-independent mouse device driver's data segment. When a complete data packet is generated, the device-dependent device driver issues an IDC Process_Packet call to the device-independent mouse device driver entry point, with the mouse device disabled. The device-independent mouse device driver performs session-dependent data processing and controls pointer image drawing.

When the device-independent mouse device driver receives a DEINSTALL request it:

1. Issues an IDC Read_Disable call to the attached device-dependent device driver
2. Performs standard deinstallation processing

When the device-dependent device driver receives a DEINSTALL request, it:

1. Issues an IDC Disable_Support call to the device-independent mouse device driver
2. Performs standard deinstallation processing (release IRQ, LID, free memory, etc.)

**Interrupt Processing:** The device-dependent device driver services all mouse device interrupts. Interrupt data is stored and converted to the common event format. The event packet is built in the shared data buffer that resides in the device-independent device driver data segment. When a packet is complete a Process_Packet call is made to the device-independent device driver to process the event. The device-dependent device driver processes the EOI for all interrupts it owns. The EOI should be issued as soon as it determines that it owns the interrupt. Before issuing the EOI, however, the mouse device should be disabled in the same manner as a Disable_Device function call. The Process_Packet call is made with the mouse device disabled. The device-independent device driver may enable the mouse device if needed with an Enable_Device function call.

The device-dependent device driver must keep track of the enable/disable state of the device. It should enable the device if needed before returning to the Interrupt Manager. Interrupts should always be disabled when returning to the Interrupt Manager.

If the device-independent device driver returns with the mouse device disabled, the device-dependent device driver should do two things. It should first disable interrupts then enable the mouse device. Interrupts should then be left disabled for the return to the Interrupt Manager.

It is important that the device-dependent device driver use the same mechanism to enable and disable the mouse device as the Enable and Disable_Device function calls do. See the device-dependent device driver IDC interface description below for more specific information on each function.

**Device-Independent Mouse Device Driver IDC Description:** The IDC provided by the device-independent device driver, MOUSE$, consists of two function calls:

- Process_Packet
- Disable_Support.

**Process_Packet Function**

This function is invoked by the attached device-dependent device driver when it has completed formatting an interrupt packet. The device-independent device driver then processes the interrupt packet. The device-dependent device driver must issue the EOI prior to calling this function, but it must leave the device disabled. This allows lower priority devices to get service and prevents the mouse device from creating uncontrolled nested interrupts. The Process_Packet function is **not** reentrant. The device-independent device driver enables and disables the mouse device as needed using the Enable/Disable_Device function calls.

**Input:**

AX = Function Request Code, Process_Packet Function = 0001H
DS = device-independent device driver's DS value
ES = device-dependent device driver's DS value (caller)
BX, CX, DX, DI, and SI register contents are undefined.

**Output:**

AX = Error Return Code, if carry set; undefined if carry clear
DS = device-independent device driver's DS value
ES = device-dependent device driver 's DS value (caller)
BX, CX, DX, SI, and DI register contents are undefined

**Note:** Error Return Codes have yet to be determined.

**Disable_Support Function**

This function is used by the device-dependent device driver to inform the device-independent device driver that it has received a Deinstall request and wishes to release its system resources. The device-independent device driver cannot deinstall itself at this point, but it can disable the API interfaces. All OS/2 mode IOCtls will return a DEVICE NOT READY error.

Input:

AX = Function Request Code, Disable_Support = 0002H
DS = device-independent device driver's DS value
ES = device-dependent device driver's DS value (caller)
BX, CX, DX, SI, and DI register contents are undefined

Output:

AX = Error Return Code, if carry set; undefined if carry clear
DS = device-independent device driver's DS value
ES = device-dependent device driver's DS value (caller)
BX, CX, DX, SI, and DI register contents are undefined

**Device-Dependent Mouse Device Driver IDC Description:** The IDC provided by the device-dependent device driver consists of five functions:

- Query_Config
- Read_Enable
- Read_Disable
- Enable_Device

- Disable_Device

**Query_Config Function:**

This function is issued by device-independent device driver during initialization to determine the configuration of the mouse device attached.

Input:

AX = Function Request Code, Query_Config Function = 0001H
DI = config data buffer offset
DS = device-dependent device driver's DS value
ES = device-independent device driver's DS value (caller)
BX, CX, DX, and SI register contents are undefined

ES:DI points to a configuration data structure in which the called device driver returns information in the following format:

```
config_data struc
   length         dw ?     ; length of structure in bytes (5), includes this field
   num_mics       db ?     ; device resolution, mickeys/centimeter
   num_butt       db ?     ; number of device buttons
   dev_irq        db ?     ; device IRQ level
   mousetype      db ?     ; Type of mouse attached
                           ; 0 = unknown
                           ; 1 = bus
                           ; 2 = serial
                           ; 3 = inport
                           ; 4 = PS/2 inboard,
                           ; 5-255 = reserved
config_data ends
```

The length field, on input, contains the number of bytes of data to be returned by the called device-dependent device driver.

Output:

AX = Error return code if carry set, undefined if carry clear
DI = config data buffer offset
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, and DS register contents are undefined

**Note:** Error Return Codes have yet to be defined.

**Read_Enable Function:**

This function is issued by the device-independent device driver to indicate that data packet processing has been setup and data transfer may begin.

Input:

AX = function request code, Read_Enable Function = 0002H
DI = shared interrupt data buffer offset
DS = device-dependent device driver's DS value
ES = device-independent device driver's DS value (caller)
BX, CX, DX, and SI register contents are undefined

ES:DI points to the interrupt data packet structure in which the called device driver will write/return the generic mouse data packet format when a complete data packet is accumulated.

```
mouse_data struc
   butt_stat        dw ?  ; standard mouse API button status flags
   col_micks        dw ?  ; relative column oriented mickey motion
   row_micks        dw ?  ; relative row oriented mickey motion
mouse_data ends
```

**Note:** Mickey motion is reported as positive if motion is to the right or up. Motion down or to the left is reported as negative.

The called device-dependent device driver should perform an AttachDD, if it hasn't already, to MOUSE$ to get the IDC entry point for issuing the Process_Packet call. In addition, this call will use a selector in the ES register. For proper bimodal support, the interrupt handling routine in the device-dependent driver should use the appropriate DS

segment/selector value returned from the AttachDD call when addressing this shared data buffer.

Output:

AX = Error return if carry set, undefined if carry clear
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, DI, and DS register contents are undefined

**Note:** Error Return Codes have yet to be defined.

**Read_Disable Function:**

This function is called by the device-independent device driver to indicate that data packet processing is no longer supported and that data transfers by way of the Process_Packet call should be discontinued until a Read_Enable request is processed. The disable is meant to be complete and no further references should be made to the shared interrupt data buffer address previously provided by the last Read_Enable request.

Input:

AX = Function Request Code, Read_Disable Function = 0003H
DS = device-dependent device driver's DS value
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, and DI register contents are undefined

Output:

AX = Error return code if carry set, undefined if carry clear
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, DI and DS register contents are undefined

**Enable_Device Function:**

This function is used to enable the mouse device interrupts for its IRQ level. In addition, for devices which require some other action to properly activate it beyond normal IRQ level manipulations, the device itself should be re-enabled and/or re-activated. This function should restore the device's operational state to that which existed prior to invocation of its counter function, Disable_Device. This call should always preserve the state of the interrupt flag.

Input:

AX = Function Request Code, Enable_Device Function = 0004H
DS = device-dependent device driver's DS value
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, and DI register contents are undefined

Output:

AX = Error return if carry set, undefined if carry clear
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, DI, and DS register contents are undefined.

**Disable_Device Function:**

This function is used to disable the mouse device interrupts for its IRQ level. The device-independent device driver has operating conditions that require the ability to specifically disable the mouse device. This function provides the ability to perform the device disable actions as well as disable the IRQ for the requested interrupt protection. The counter function, Enable_Device, must be used to awaken the device and activate the IRQ level.

Input:

AX = Function Request Code, Disable_Device Function = 0005H
DS = device-dependent device driver's DS value
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, and DI register contents are undefined

Output:

AX = Error return if carry set, undefined if carry clear
ES = device-independent device driver's DS value (caller)
BX, CX, DX, SI, DI, and DS register contents are undefined

## OS/2 Mode Mouse Support

This section describes the standard OS/2 mouse device support for OS/2 mode applications.

**Overview:** Mouse device drivers have characteristics which are quite different from most other devices. They are read-only devices which provide data at approximately 30 events per second. The data is structured, that is, it will arrive as a packet of data containing absolute screen location, button up/down and other information.

The OS/2 mode mouse driver model described in this section is designed to provide a basic, machine-independent, high-performance interface. Applications and Environment Managers may use this interface to obtain mouse device services.

The Base Mouse Subsystem (BMS) is a dynamic link module which executes on level 3 (application level). The BMS receives all Mou*Xxx* calls (as a common entry point) and passes those calls to the appropriate handler. One handler for each session is allowed.

Normally, the system supplied default handler is the session's mouse handler. However, Environment Managers, OEM, and custom mouse device drivers may find it necessary to intercept Mou*Xxx* calls for various reasons. A particular custom handler may service many different sessions, provided it uses MouRegister with each of those sessions.

Pointer image updating is executed by the mouse device driver utilizing functions supplied by the display device drivers. The pointer updating occurs at interrupt time, ensuring that the pointer shape moves smoothly and promptly across the screen. The Mou*Xxx* API contains three commands which allow the application to:

- Set the pointer shape
- Reserve a collision area where the pointer must not be drawn
- Free a collision area to the pointer device.

The responsibilities of the mouse driver have been well separated from those of a screen driver. Pointer updating functions call the display device drivers rather than attempting to draw the pointer image directly. This maintains independence between the mouse and display devices. It does require that custom display device drivers conform to the pointer device driver interface to allow the pointer shape to be drawn and also requires that the application synchronize display access.

**Pointer Draw Installation:** The mouse pointer image update design allows the update routine to be installed with the video subsystem. Custom and OEM video subsystems, which allow display modes not supported by the base video subsystem, may implement interrupt-time screen pointer image updating by providing screen image draw routines for execution by the pointer device driver.

Screen pointer draw routines are called by the mouse pointer device driver at interrupt time. The screen pointer draw routines are installed as character device drivers at IPL time.

The necessary steps for installing a screen pointer image draw routine are outlined below:

- Pointer draw routines are installed at IPL time by including them on a DEVICE= keyword in the CONFIG.SYS file as named character device drivers. No special mechanism is needed. The default pointer draw device driver file named POINTDD.SYS is needed.
- The display (screen) =IOCtl (category 3, Function 72H) must be supported by the named pointer draw device driver. This IOCtl allows the mouse subsystem router/handler to query the pointer draw device driver for the entry (far call) address.
- When an application uses MouOpen to a mouse handle, the mouse subsystem handler/router will inspect the stack to determine if the call specified a non-system pointer draw device driver name.
  - If the pointer is 0, the mouse subsystem will use the default (system supplied) device driver.

- If the pointer was not 0, the mouse subsystem will follow the pointer to get the ASCIIZ name of the pointer draw device driver.

- The mouse subsystem handler/router will OPEN the pointer draw character device driver using DosOpen. Using the device handle returned by the OPEN, the handler/router will then issue the category 3, Function 72H screen IOCtl to get the entry address (selector : offset) of the pointer draw device driver. The mouse device driver will issue FAR calls to this entry address when the mouse has a pointer manipulation request.
- The mouse subsystem handler/router will then do a DosOpen to the mouse device driver. The return value from this call will be a standard OS/2 device handle.
- The mouse subsystem handler/router will use the DosOpen device handle to pass the entry address of the pointer draw routine to the mouse device driver. This is done by using the category 7, Function 5AH mouse IOCtl addressed to the mouse device driver's DOS handle.
- The mouse device driver will call the pointer draw routine with each request without their being linked prior to start time. Linkage will be established (through the IOCtls) for each pointer draw device driver specified by a MouOpen.
- There may be only one pointer draw routine (driver) for each session.
- This mechanism applies to OS/2 mode only.

**Handler/Router:** There are three aspects to an OS/2 mouse handler/router:

- Mou*Xxx* API to allow applications to avoid the specifics of the low level IOCtl interface
- Circular buffers to receive events
- Pointer management interface.

The driver is installed as a character device, with the name "MOUSE*xxx*.SYS." The MouOpen function call initializes the device (sets initial coordinates, checks for mouse presence).

The Mou*Xxx* interface allows the caller to obtain information about the current state of the mouse, set parameters, allow the application to determine which events are to be passed into the device circular buffer, and others.

The circular I/O buffer is a high-efficiency buffer shared by all client applications in the session and the mouse device driver. There is only one queue for each session, no matter how many applications within the session are utilizing the mouse device. The driver uses this interface to provide "events." The caller can specify what constitutes an event. Examples of events are pressing buttons and moving a mouse.

Events are time-stamped so that a higher-level interface library package can provide such features as pressing the mouse button twice.

**Coordinates:** Coordinates are mouse event's "absolute" position relative to the top left corner (0,0) of the display screen. This means that the units in which the mouse position is reported depends on the display mode in which the session is executing. There are two different possibilities:

- In TEXT mode, pointer position is reported in CHARACTER units.
- In GRAPHICS mode, pointer position is reported in PIXELS.

By supplying pointer coordinates as offsets to absolute screen position, higher level library support for translating data into an absolute coordinates may no longer be necessary.

For those systems that wish to operate in terms of "relative" mouse movement (mickey) displacements, the MouSetDevStatus call allows the library support or the application to set the mouse device driver to return mickey movements and not screen coordinates.

**Motion:** The unit of motion for a mouse is known as a "mickey." This is similar to the pixel, the unit of addressability on a screen.

The OS/2 mouse driver provides calls to determine the number of units of motion per centimeter, so that an application, window manager or other package can relate motion to a physical screen.

**Mou *Xxx*** and IOCtl Calls: The mouse IOCtl is category 7. Applications should not concern themselves with the details of the mouse IOCtl interface. Instead, the OS/2 Mouse device driver should be accessed by applications through the Mou*Xxx* API. Only Environment Managers and custom mouse device drivers need be aware of the IOCtl interface.

All mouse device driver IOCtl functions have an application-level MOU API equivalent function. The IOCtl function codes and their MOU API equivalents are as follows:

Table 6-3. Mouse Device Driver IOCtl Functions

| IOCtl Fcn | MOU API Function | Function Performed . |
|---|---|---|
| IOMR_NB | MouGetNumButtons | Get # of mouse buttons |
| IOMR_MC | MouGetNumMickeys | Get # of mickeys/centimeter |
| IOMR_GS | MouGetDevStatus | Get device status flags |
| IOMW_DS | MouSetDevStatus | Set device status flags |
| IOMR_QS | MouGetNumQueEl | Get event queue status |
| IOMR_RD | MouReadEventQue | Read event queue contents |
| IOMR_GF | MouGetScaleFact | Get current scaling factors |
| IOMW_SS | MouSetScaleFact | Set new scaling factors |
| IOMR_GM | MouGetEventMask | Get current event mask |
| IOMW_EM | MouSetEventMask | Set new event mask |
| N/A | MouOpen | Open mouse support |
| N/A | MouClose | Close mouse support |
| N/A | MouRegister | Install a mouse subsystem |
| N/A | MouDeRegister | Deinstall a mouse subsystem |
| N/A | MouInitReal | Initialize DOS mode driver |
| IOMW_SP | MouSetPtrShape | Assign new pointer shape |
| IOMW_GP | MouGetPtrShape | Assign new pointer shape |
| IOMW_DP | MouDrawPtr | Unmark collision area |
| IOMW_RP | MouRemovePtr | Mark collision area |

## OS/2 Mode Mouse API

Refer to the *OS/2 Version 1.2 Programming Reference* and the *OS/2 Version 1.2 Programming Guide* for a discussion of the OS/2 mode Mouse function calls. A summary of Mouse OS/2 mode API descriptions follows:

**MouRegister** Register mouse subsystem
**MouDeRegister** Deregister mouse subsystem
**MouInitReal** Initialize DOS mode pointer draw
**MouOpen** Opens the mouse device for the current session
**MouClose** Closes the mouse device for the current session.
**MouDrawPtr** Release screen area for device driver use
**MouRemovePtr** Reserve screen area for application use
**MouFlushQue** Flush mouse event queue
**MouGetDevStatus** Query current pointing device driver status flags
**MouGetEventMask** Query current pointing device one-word event mask
**MouGetNumButtons** Query number of buttons
**MouGetNumMickeys** Query number of mickeys per centimeter
**MouGetNumQueEl** Query current status for the pointing device event queue
**MouGetPtrPos** Query current pointer position
**MouGetPtrShape** Query pointer shape and size
**MouGetScaleFact** Query scale factors for the current pointing device
**MouReadEventQue** Read the pointing device event queue

| | |
|---|---|
| **MouSetDevStatus** | Set device status flags |
| **MouSetEventMask** | Assign new event mask to the current pointing device |
| **MouSetPtrPos** | Set current pointer position |
| **MouSetPtrShape** | Set pointer shape and size |
| **MouSetScaleFact** | Set scale factors for the current pointing device |
| **MouSynch** | Synchronize (serialize) access to the mouse device driver |

The MouOpen and MouClose commands are mapped by the Mouse Base Subsystem Router into the DosOpen and DosClose system calls, respectively. MouRegister and MouDeRegister are directed to the higher level Mouse Base Subsystem Router and do not translate down to the IOCtl device driver level.

The Base Mouse Subsystem receives all Mou*Xxx* function calls. Function specific data is passed on the user's stack in the following format:

```
Mouse Router return address    2 words <== Top of Stack
Value of application DS reg.   1 word
Entry point return address     1 word
Function code                  1 word
Application (caller) ret addr  2 words
Function specific parameters   2 - 10 bytes depending on call
```

**Events:** The mouse driver provides data to the user through the standard OS/2 asynchronous and parallel I/O interfaces described in other sections of this Chapter.

Mouse events are placed in a circular I/O buffer. The conditions which generate an event are controllable through the event mask feature and are available by way of the MouSetEventMask call or the equivalent mouse IOCtl command.

Mouse events have the following format:

```
WORD    Event Mask
DWORD   time stamp in milliseconds
WORD    Row absolute / mickeys
WORD    Column absolute / mickeys
```

These fields have the following meaning:

**Event Mask:** This indicates which events are in this record. See the MouGetEventMask call description in the *OS/2 Version 1.2 Programming Reference* for details on the event mask bit definitions.

**Time in Milliseconds:** This is a time stamp for the event. It is provided so that higher level interface packages can provide features like two mouse button presses with selective timing, and so that mouse and keyboard monitor events can be synchronized. The time value is the number of milliseconds since the last IPL.

**Row Absolute and Column Absolute / Mickeys:** This may indicate either absolute position of the mouse pointer shape relative to the top left corner of the display screen or mouse movement (mickey).

If reporting coordinates (the default), they will be in either pixel or character offsets, depending on whether the display mode for the session is graphics or text, respectively.

The application must explicitly invoke reporting of mouse mickey units with the MouSetDevStatus call. In this case, events are reported in units of mouse movement. For rows, a negative number means movement toward the upper part of the screen. For columns, a negative number means movement toward the left part of the screen.

**Pointer:** Maintenance of the pointer (shape and location) is performed by the mouse device driver. An application must provide the mouse driver with a pointer image that the driver will use to draw the pointer for that session.

Applications utilizing mouse services need to ensure that the mouse device driver and the application do not attempt to update the screen at the same location, at the same time. The MOU API provides three commands to accomplish these functions:

- MouSetPtrShape
- MouDrawPtr
- MouRemovePtr

It is the responsibility of the application to synchronize pointer operations between itself and the pointer shape draw routine.

**Display Modes Supported:** Graphic modes are only supported in the OS/2 mode if the application performs all drawing and moving of the pointer. The controlling application must first issue MouSetDevStatus and indicate no interrupt time pointer drawing, plus pointer movement reported in mickeys. Next, issue VioSetMode to the desired graphics modes. The graphic modes that are supported are 4, 5, 6, 0DH, 0EH, 010H, 011H, 012H, 013H and the 8514/A adapter Advanced Functions modes.

## Mouse Monitors

Some applications need to view mouse device events as they arrive from the device driver. These applications may wish to consume some of the mouse events, or they may wish to replace some mouse events with one or more other mouse events. This is made possible by the "mouse monitor" function.

When the DosMonReg call is used with mouse devices, the index indicates the session, from 0 to 15 (see DosGetInfoSeg). -1 indicates the session of the calling thread.

The size of the mouse device driver's data buffer is 16 bytes. This is the value to be used in calculating the sizes of the input/output buffers required for the DosMonReg call.

The mouse device driver supports device monitors. The device driver passes information to the monitor in packets consisting of a word of monitor flags plus the standard mouse device driver event buffer. The packet format is as described below:

```
WORD  <00> -- Monitor Flags -- open, close, etc.
WORD  <02> -- Event mask (see MouGetEventMask for definitions)
DWORD <04> -- Time stamp in milliseconds
WORD  <08> -- Absolute horizontal (x or row) screen position
WORD  <0A> -- Absolute vertical (y or column) screen position
```

The following DevHlp monitor control functions are used by the OS/2 Mouse Device drivers to implement mouse monitors:

- MonitorCreate
- Register
- Deregister
- MonWrite
- MonFlush.

The OS/2 Mouse Device Driver uses the DevHlp monitor control functions for the following situations:

**MonitorCreate** Create a monitor chain for each OS/2 session, supported by the system, when the Mouse Device Driver receives and processes the system's INIT request.

Remove each support session's monitor chain when a DEINSTALL request is received and processed.

**Register** Add a monitor to a monitor chain for the current session when a register request is received and processed.

**Deregister** Remove a monitor from the current session's monitor chain when a deregister request is received and processed.

**MonWrite** Provide the current session's monitor chain event data at interrupt time.

**MonFlush** Flush all data from session's monitor chain.

See Chapter 8, "Character Device Monitors" for more information.

## DOS Mode Mouse Support

This section describes the standard OS/2 mouse device support for DOS mode applications.

**Overview:** DOS mode mouse support is based on the Microsoft INT 33H support.

DOS mode Mouse support preserves the INT 33H interface and is substantially compatible with existing Microsoft support.

**Pointer Draw Installation:** DOS mode pointer draw installation is implemented through the MouInitReal call. The shell issues the MouInitReal call during shell initialization. The sequence of events is as follows:

At SYSINIT (IPL) time, the shell (System Session Manager) issues MouInitReal.

In order to establish addressability between the DOS mode mouse device driver and the DOS mode screen pointer draw routine:

- The shell issues a MouInitReal to open the system default Pointer Draw Device Driver.
- The MouInitReal issues category 3 (screen) IOCtl 72H to the DOS mode/OS/2 mode (shared) mode screen pointer draw device driver. This IOCtl will return the address of the pointer draw routine entry point.
- The MouInitReal will then issue category 7 (mouse) IOCtl 5BH to the DOS mode mouse device driver. This IOCtl passes the pointer draw address obtained from the preceding category 3, function 72H IOCtl to the mouse device driver.
- MouInitReal will then return to the shell with a completion code indicating the result of the DOS mode mouse initialization process.

**Handler/Router:** The Mouse Handler/Router is applicable only to OS/2 mode mouse support. In OS/2 mode, it directs operations among multiple sessions. In addition, it allows the Mou*Xxx* calls to be intercepted and synchronized between multiple processes in a session. This interception is done by an environment Manager, mouse subsystem, or a sophisticated application.

DOS mode mouse support calls may be intercepted by hooking the INT 33H vector. While there are multiple OS/2 mode sessions allowed, there may not be more than one DOS mode session. Consequently, there is no need for, and no support provided for, a DOS mode mouse handler/router.

**Coordinates:** The OS/2 mode mouse reports its coordinate position in absolute displacement (characters or pixels) from the upper left corner of the screen. In a similar manner, DOS mode mouse support reports mouse coordinates relative to the upper left corner of the screen.

In contrast, the DOS mode mouse support reports the position in virtual screen units. The virtual display coordinates are relative to the display mode. Refer to the table on page 6-36 for a list of the supported display modes and their virtual display coordinates.

The OS/2 Mouse Device Drivers do not limit the virtual display coordinate settings. However, if an application wishes to define the display with a large virtual coordinate grid than the physical, it should also be prepared to perform the pointer image drawing because the OS/2 virtual coordinate support will always map back to the virtual display space. The relative displacement of one unit will depend on the dimensions of the screen and the associated resolution for the mode setting on the display while the DOS mode is the foreground session.

**Motion:** As with OS/2 mode mouse support, the unit of motion for a mouse is known as a "mickey". This is similar to the pixel, the unit of addressability on a screen.

**IOCtl Calls:** There is only one IOCtl supported for the DOS mode mouse device driver.

Portions of the mouse device driver are used by both the DOS mode and OS/2 mode device drivers. This shared portion supports the Category 7 (mouse), Function 5BH IOCtl for the DOS mode. This IOCtl is only used by the shell and then only at shell initialization time.

The OS/2 mode shell (System Session Manager) always exists in the OS/2 system. It is the shell (at shell initialization time) that calls for DOS mode mouse device initialization. The purpose of this

shell call is to determine the entry point of the screen pointer draw device driver. It is this entry point which the mouse device driver will call on interrupts to have the pointer image updated for the DOS mode session.

There are no other mouse IOCtls in OS/2 that are supported by the DOS mode device driver.

**Events:** The Microsoft INT 33H DOS mode mouse API is designed to detect and report "changes" in the state of the mouse. Consequently, the mouse support reports events such as:

- Button Press data
- Button Release data
- Button Number
- Mouse movement.

**Pointer:** The DOS mode mouse API supports application pointer image setting. Two commands are provided to enable the application to modify the DOS mode pointer shape. They are:

- Set Text Pointer Shape
- Set Graphics Pointer Shape.

The first command is used while the display is in text modes. The second command is used while the display is in graphics modes.

**Display Modes Supported:** All text modes are supported (0, 1, 2, 3 and 7). Graphic modes 4, 5, 6, 0DH, 0EH, 0FH and 010H are also supported.

## DOS Mode INT 33H Mouse API

Please refer to the *OS/2 Programming Guide* for a discussion of the function calls dealing with Mouse OS/2 mode APIs. OS/2 supports a subset of the Microsoft DOS INT 33H mouse API. Refer to the table on page 6-36 for the display modes supported.

The Microsoft INT 33H mouse API is available to only those applications executing in the DOS mode. OS/2 mode applications must use the MOU*xxx* mouse device interface.

The Mouse Device Driver provides DOS mode applications with an INT 33H interface to the pointing device hardware. The DOS mode support is not equivalent to the OS/2 OS/2 mode support instead, it preserves the Microsoft INT 33H mouse interface.

The DOS mode Mouse does not support:

- Device handles
- Monitor chains
- IOCtl direct function access
- MOU API function calls.

All DOS mode mouse functions are accessed on the software INT 33H interface. When a software interrupt 33H is detected, a DOS mode Mouse Handler Routine is invoked. All function relevant information is supplied by the caller in the following seven registers:

- AX
- BX
- CX
- DX
- SI
- ES
- DI.

There are two states for the DOS mode Mouse support:

- The OS/2 Mouse Device Driver is loaded into the operating system. When this state is active, all DOS mode Mouse functions listed in this section are available for user support.
- The OS/2 Mouse Device Driver processes a DeInstallation request. In this state the DOS mode Mouse support is limited to INT 33H, Function call 0. This is also true if an invalid/unsupported display mode was set.

**Mouse Device Driver Interfaces/Requirements:** To get DOS mode mouse support, use MODE=R or MODE=B on the DEVICE= statement of CONFIG.SYS.

**Mouse Device Driver Button Definitions:** The DOS mode button definition depends on the number of buttons on the mouse device.

Button definitions are used on functions 3, 5, 6, and 12. Button number assignments are as follows:

**Two-button mouse:**

| *Bit#* | *Mouse Button* |
|---|---|
| 1 | Rightmost button |
| 0 | Leftmost button |

**Three-button mouse**

| *Bit#* | *Mouse Button* |
|---|---|
| 2 | Center button |
| 1 | Rightmost button |
| 0 | Leftmost button |

**Mouse Device Driver Function Summary:** The software INT 33H interface is provided only for DOS mode support. The INT 33H Mouse API provides the following functions:

| # | *Function Performed* |
|---|---|
| 0 | Mouse Installed Flag and Reset |
| 1 | Show Mouse Pointer |
| 2 | Hide Mouse Pointer |
| 3 | Get Mouse Pointer Position & Button Status |
| 4 | Set Mouse Pointer Position |
| 5 | Get Button Press Information |
| 6 | Get Button Release Information |
| 7 | Set Min and Max Horizontal Position |
| 8 | Set Min and Max Vertical Position |
| 9 | Set Graphics Pointer Shape |
| 10 | Set Text Pointer Shape |
| 11 | Read Mouse Motion Counters |
| 12 | Set User-Defined Subroutine Input Mask |
| 13 | Light Pen Emulation Mode ON |
| 14 | Light Pen Emulation Mode OFF |
| 15 | Set Mickey/Pixel Ratio |
| 16 | Conditional OFF |
| 19 | Set Double Speed Threshold |
| 20 | Swap User-defined Subroutine |
| 21 | Query Save Mouse State Storage Requirements |
| 22 | Save Mouse Driver State |
| 23 | Restore Mouse Driver State |

The function number and the function specific parameters are passed to the Mouse Device Driver in the four general purpose registers and the SI, DI and ES registers.

- The AX register is always used to contain the requested function number
- The BX, CX, and DX registers are used as needed for function-specific input parameters
- SI and DI are used for function call 16
- ES is used for function calls 9 and 12.

On return from a software interrupt 33H function call, the general purpose registers contain return codes and/or mouse requested data items. The registers used are function-specific and detailed under each individual call description.

All input parameters are checked on function calls requiring parameters.

If an INT 33H function call is issued when the Mouse Hardware/Software is not properly initialized, the call will return an error code of 0 in AX.

The INT 33H Interface is based on the concept of a virtual display screen coordinates. All coordinates are input/output relative to a default range of rows and columns. The default virtual range is required for the OS/2 Mouse Device Driver to perform the pointer image tracking on the display. However, an application is not stopped from altering the virtual display coordinate limits through the INT 33H interface. If an application alters virtual display coordinate grids to be greater than the physical display resolution, the pointer tracking will be unpredictable and all virtual coordinates read from the INT 33H interface may be inaccurate. This is properly supportable if the application which set the virtual coordinate ranges also assumes responsibility of pointer image drawing. The application can assume the INT 33H motion counters will not be affected and therefore is a stable source of pointer image motion data.

The mouse device driver maps the physical display resolution to the virtual display screen coordinate system independent of the physical display mode.

## INT 33H-0 Mouse Installed Flag and Reset

**Purpose** Determines if the DOS mode Mouse device is present.

If the appropriate mouse device hardware and software are available, this function will return a value of -1 in the AX register. In addition, this call will set/reset all of the software tracking mechanisms used by the other DOS mode mouse function calls to their default values.

If the mouse hardware and/or software is not available for interaction, or if the video mode is not supported, this call will return a value of 0 in AX.

**Input Parameters:**

AX = function code of 0
BX = not used
CX = not used
DX = not used

**Return codes/data values:**

If mouse support is not available, then
  AX = 0
  If mouse support is installed then
    BX = -1
else (mouse support is available)
  AX = -1
  BX = number of mouse device buttons supported.

The following table defines the default function values set when Mouse support is available (AX = -1):

| *Function* | *Default value* |
|---|---|
| Mouse pointer position | Screen center |
| Pointer redraw flag | -1, Pointer hidden |
| Graphics pointer image | Mouse default image |
| Text pointer image | Reverse video |
| Interrupt call mask | All 0's, no routine in use |
| Light pen emulation mode | Enabled |
| Mickey/pixel ratio (horizontal) | 8 to 8 |
| Mickey/pixel ratio (vertical) | 16 to 8 |
| Min/Max ptr position (horizontal) | Display/mode dependent |
| Min/Max ptr position (vertical) | Display/mode dependent |

## INT 33H-1 Show Mouse Pointer

**Purpose** Requests that the mouse device driver draw the pointer image and update the pointer redraw flag.

This function determines if the mouse pointer image is already visible (pointer redraw flag = 0). If the pointer image is already visible, this function returns to the caller.

If the pointer redraw flag is not 0, this call increments the pointer redraw flag. After the pointer redraw flag is incremented, it is checked again to see if it is equal to 0.

If the redraw flag was incremented to 0, the pointer image is redrawn on the display screen.

If the internal cursor flag is already 0, this function has no effect.

This function also clears conditional off (function 16) values.

**Input Parameters:**

AX = function code of 1
BX = not used
CX = not used
DX = not used

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-2 Hide Mouse Pointer

**Purpose** Requests the mouse device driver to hide the pointer image and update the pointer redraw flag.

This function decrements the mouse pointer redraw flag. If the pointer image is already hidden, the call returns to the application. If the mouse pointer redraw flag was decremented to -1, the pointer image is hidden.

When the cursor is hidden, the mouse device driver continues to track the motion of the pointer on the screen. It simply does not draw the pointer image on the display.

This function is used to ensure that the pointer device driver will not interfere with data being written on the screen by application programs.

This function always decrements the cursor flag regardless of its current value.

**Input Parameters:**

AX = function code of 2
BX = not used
CX = not used
DX = not used

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-3 Get Mouse Pointer Position & Button Status

**Purpose** Returns the current mouse button status and pointer image screen coordinates ((columns, rows) or (x, y)).

**Input Parameters:**

AX = function code of 3
BX = not used
CX = not used
DX = not used

**Return codes/data values:**

AX = Unchanged
BX = mouse button status
CX = horizontal (x or column) pointer coordinate
DX = vertical (y or row) pointer coordinate

The mouse button status value is a bit mask indicating which mouse buttons are currently pressed/released.

A SET bit is defined as a button being pressed.

| *Bit#* | *Value/meaning* |
|---|---|
| 3-15 | Reserved = 0 |

**For two button mouse**

| *Bit#* | *Value/meaning* |
|---|---|
| 2 | Reserved = 0 |
| 1 | Set if rightmost button pressed |
| 0 | Set if leftmost button pressed |

**For three button mouse**

| *Bit#* | *Value/meaning* |
|---|---|
| 2 | Set if center button pressed |
| 1 | Set if rightmost button pressed |
| 0 | Set if leftmost button pressed |

The pointer coordinates are relative to the range defined for the mouse under the virtual terminal concept. The ranges are display mode dependent. See "Mouse Screen Resolutions" on page 6-36 for a description of the virtual screen resolution by mode.

## INT 33H-4 Set Mouse Pointer Position

**Purpose** Assigns the mouse pointer image to a new screen location.

If the coordinates are too large, or too small, the mouse pointer position is set to the maximum or minimum screen values, respectively.

**Input Parameters:**

AX = function code of 4
BX = not used
CX = new pointer horizontal coordinate
DX = new pointer vertical coordinate

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-5 Get Button Press Information

**Purpose** Returns a specified button's status information.

Button status information consists of:

- Up/down state of all buttons
- A button press counter value
- The last button press screen position coordinates.

The button press counter contains the number of times the requested button was pressed since the last time this call was issued.

The button press counter is reset to 0 by this call. There is no overflow checking performed for the button press counters when they are updated.

The button press screen position coordinates are always reported in a virtual display mode value.

The pointer coordinates are relative to the range defined for the mouse under the virtual terminal concept. The ranges are display mode dependent. See "Mouse Screen Resolutions" on page 6-36 for a description of the virtual screen resolution by mode. The values follow:

**Input Parameters:**

AX = function code of 5
BX = button status requested
0 = leftmost button

1 = rightmost button
2 = center button
CX = not used
DX = not used

**Return codes/data values:**

AX = Bit-mapped as follows:

| *Bit#* | *Meaning* |
|---|---|
| 0 | = 0 If leftmost button is UP, 1 if down |
| 1 | = 0 If rightmost button is UP, 1 if down |
| 2 | = 0 If center button is UP, 1 if down |
| 3-15 | = Not used |

BX = Button counter value
CX = Last button press horizontal coordinate position
DX = Last button press vertical coordinate position

This function uses AX to return a value. If the input parameter in BX is illegal, then the output registers are returned as:

AX = 0
BX = 0
CX = 0
DX = 0

## INT 33H-6 Get Button Release Information

**Purpose** Returns a specified button's status information.

Button status information consists of:

- Up/down state of all buttons
- A button release counter value
- The last button release screen position coordinates.

The button release counter will contain the number of times the requested button was released since the last time this call was issued.

The button release screen position coordinates are always reported in a virtual display mode value.

The pointer coordinates are relative to the range defined for the mouse under the virtual terminal concept. The ranges are display mode dependent. See "Mouse Screen Resolutions" on page 6-36 for a description of the virtual screen resolution by mode. The returned values are:

**Input Parameters:**

AX = function code of 6
BX = button status requested
0 = leftmost button
1 = rightmost button
2 = center button
CX = not used
DX = not used

**Return codes/data values:**

AX = Bit-mapped as follows:

| *Bit#* | *Meaning* |
|---|---|
| 0 | = 0 If leftmost button is UP, 1 if down |
| 1 | = 0 If rightmost button is UP, 1 if down |
| 2 | = 0 If center button is UP, 1 if down |
| 3-15 | = Not used |

BX = Button counter value
CX = Last button press horizontal coordinate position
DX = Last button press vertical coordinate position

This function uses AX to return a value. If the input parameter in BX is illegal, then the input registers are returned as:

AX = 0
BX = 0
CX = 0
DX = 0

## INT 33H-7 Set Min & Max Horiz Position

**Purpose** Assigns virtual screen minimum and maximum horizontal coordinate positions.

By defining virtual screen horizontal minimum and maximum coordinates, the pointer image is limited to a subset of the physical display horizontal movement area.

If the value is too small, the current minimum value is used. Values larger than the maximum, the physical display resolution, may be used but with unpredictable pointer image tracking results.

If the maximum value is less than the minimum, then the two values are swapped.

If the pointer image is outside of the area when the call is made, it is moved to just inside the area.

**Input Parameters:**

AX = function code of 7
BX = not used
CX = minimum virtual screen horizontal position
DX = maximum virtual screen horizontal position

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-8 Set Min & Max Vertical Position

**Purpose** Assigns virtual screen minimum and maximum vertical coordinate positions.

By defining virtual screen vertical minimum and maximum coordinates, the pointer image is limited to a subset of the physical display vertical movement area.

If the maximum value is less than the minimum, the two values are swapped.

If the pointer image is outside of the area when the call is made, it is moved to just inside the area.

If a superset of the physical display is defined in the call, the maximum will be defined off the display. In this case the pointer image will not travel off the edge of the display even though the reported pointer coordinates will continue to track out to the defined virtual boundary. An application could perform this extended mapped pointer drawing itself through other INT 33H Functions.

Applications that change the display modes' default font through INT 10H must also issue this function call to remap the entire physical display space into the Virtual Display space.

**Input Parameters:**

AX = function code of 8
BX = not used
CX = minimum virtual screen vertical position
DX = maximum virtual screen vertical position

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-9 Set Graphics Pointer Shape

**Purpose** Assigns a new graphics mouse pointer image.

This function defines the shape, color and hot spot of the mouse pointer when the display is in graphics mode.

The following pointer image information must be provided:

- New pointer image horizontal hot spot coordinate
- New pointer image vertical hot spot coordinate
- Pointer to the new pointer image buffer.

The hot spot coordinates are pixel value indices relative to the upper left corner of the pointer. The relative hot spot coordinates must be in the range of ± 16.

The pointer image buffer must have a length of 64 bytes. The pointer image buffer is logically divided into two bit level masks:

- The first 32 bytes define the screen mask. The screen mask determines whether the pointer pixels are part of the shape or background.
- The last 32 bytes define the pointer mask. The pointer mask determines how the pixels under the pointer contribute to the color of the pointer.

The pointer draw routine first logically ANDs the screen mask with the 256 bits of data that define pixels under the pointer. Then it logically XORs the pointer mask with the result of the AND operation.

In modes 6, ODH, OEH, OFH and 010H each screen bit defines the color of a single pixel. Thus, one bit in the screen mask and one bit in the pointer mask define the pixel's color when the pointer is over it.

In modes 4 and 5, each pair of screen bits defines the color of a single pixel. Consequently, a pair of bits in the screen mask and a pair in the pointer mask define a pixel's color.

**Input Parameters:**

AX = function code of 9
BX = pointer hot spot (horizontal position)
CX = pointer hot spot (vertical position)
DX = address of screen and pointer masks
ES = segment of screen and pointer masks

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

## INT 33H-10 Set Text Pointer Shape

**Purpose** Defines a text (character) pointer image.

**Input Parameters:**

AX = Function code of 10
BX = pointer select
    0 selects the software text pointer
    1 selects the hardware cursor
CX = Screen mask value/hardware cursor start scan line
DX = Pointer mask value/hardware cursor stop scan line

For the software text pointer, the masks (CX and DX) are bitmapped as follows:

| *Bit#* | *Meaning* |
|---|---|
| 15 | Blinking |
| 14-12 | Background Color |
| 11 | Intensity |
| 10-8 | Foreground Color |

7-0 Character

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-11 Read Mouse Motion Counters

**Purpose** Returns the number of mickeys the mouse has moved horizontally and vertically since the last time this function was called.

The returned value is between -32768 and 32767.

A positive number indicates motion to the right for horizontal motion, and to the bottom for vertical motion.

This call sets the counts to 0. Overflow is ignored.

**Input Parameters:**

AX = Function code of 11
BX = Not used
CX = Not used
DX = Not used

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Horizontal count
DX = Vertical count

## INT 33H-12 Set User-Defined Subroutine Input Mask

**Purpose** Sets the call mask and subroutine address for the mouse hardware interrupts.

The mouse driver will call the designated subroutine if any of the mask conditions are met.

To cause the subroutine to be invoked for a certain condition, set the corresponding bit in the call mask to a 1. If the subroutine is not to be invoked for a condition, the corresponding bit should be a 0.

**Input Parameters:**

AX = Function code of 12
BX = Not used
CX = Call mask
DX = Offset of subroutine
ES = Segment of subroutine

The call mask is a word value with the following bit map:

| *Bit#* | *Meaning* |
|---|---|
| 15-7 | Reserved (0) |
| 6 | Center button released |
| 5 | Center button pressed |
| 4 | Rightmost button released |
| 3 | Rightmost button pressed |
| 2 | Leftmost button released |
| 1 | Leftmost button pressed |
| 0 | Pointer position changed |

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

When the mouse driver calls the subroutine, it loads the following values into the general purpose registers:

AX = Condition mask (similar to the call mask except a bit is set only if the condition has occurred.)
BX = Button State
CX = Pointer Coordinate (horizontal)
DX = Pointer Coordinate (vertical)
SI = Last raw vertical mickey count read from mouse
DI = Last raw horizontal mickey count read from mouse.

**Note:** Because the DS register contains the mouse driver data segment, the user's subroutine must set it to its own data segment value.

## INT 33H-13 Light Pen Emulation Mode ON

**Purpose** Instructs the mouse device driver to emulate a light pen.

Calls to the PEN function in IBM Basic will return the pointer position at the last "pen down."

The "pen down" state is created by pressing the leftmost and rightmost buttons at the same time.

The "pen off the screen" state occurs when either button is up.

**Input Parameters:**

AX = Function code of 13
BX = Not used
CX = Not used
DX = Not used

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-14 Light Pen Emulation Mode OFF

**Purpose** Disables the mouse device driver light pen emulation.

**Input Parameters:**

AX = Function code of 14
BX = Not used
CX = Not used
DX = Not used

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-15 Set Mickey/Pixel Ratio

**Purpose** Sets the mickey-to-pixel ratio for horizontal and vertical mouse motion.

The ratios specify the number of mickeys per eight pixels. The values must be in the range:

1 < = value < = 32767.

The ratio is used to determine the amount of pointer image movement to be associated with physical mouse motion. The ratio is specified in the number of mickeys per eight pixels. For example, if a value of six is set, then if the mouse device is moved a distance of six mickeys the pointer image will move eight pixels.

**Input Parameters:**

AX = Function code of 15
BX = Not used
CX = Horizontal Mickey/Pixel Ratio
DX = Vertical Mickey/Pixel Ratio

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-16 Conditional Off

**Purpose** Defines a region on the screen for updating.

If the mouse pointer is in the defined region, or moves into it, this function will hide the defined region while it is being updated. After this function is called, function 1 must be called later to show the pointer again.

This function is similar to function 2, but is intended for advanced applications that need quicker screen updates. Because of the number of parameters required, this function cannot be called from interpreted BASIC.

**Input Parameters:**

AX = Function code of 16
BX = Not used
CX = Left Column (x or width) screen coordinate
DX = Upper Row (y or height) screen coordinate
SI = Right Column (x or width) screen coordinate
DI = Lower Row (y or height) screen coordinate

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-19 Set Double Speed Threshold

**Purpose** Sets the threshold speed for doubling pointer motion on the screen.

The default value is 128 mickeys per second. If the mouse moves faster than this number, pointer motion doubles in speed.

**Input Parameters:**

AX = Function code of 19
BX = Not used
CX = Not used
DX = Threshold speed in mickeys/second

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

## INT 33H-20 Swap User-Defined Subroutine

**Purpose** Sets the call mask and subroutine address for the mouse hardware interrupts and returns the previous values of the call mask and subroutine address.

The mouse driver will call the designated subroutine if any of the mask conditions are met.

To cause the subroutine to be invoked for a certain condition, set the corresponding bit in the call mask to a 1. If the subroutine is not to be invoked for a condition, the corresponding bit should be a 0.

**Input Parameters:**

AX = Function code of 20
BX = Not used
CX = Call mask
DX = Offset of subroutine
ES = Segment of subroutine

The call mask is a word value with the following bit map:

| *Bit#* | *Meaning* |
|---|---|
| 15-7 | Reserved (0) |
| 6 | Center button released |
| 5 | Center button pressed |
| 4 | Rightmost button released |
| 3 | Rightmost button pressed |
| 2 | Leftmost button released |
| 1 | Leftmost button pressed |
| 0 | Pointer position changed |

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Previous call mask
DX = Offset of previous subroutine
ES = Segment of previous subroutine

When the mouse driver calls the subroutine, it loads the following values into the general purpose registers:

**AX** = Condition mask (Similar to the call mask except a bit is set only if the condition has occurred.)
**BX** = Button State
**CX** = Pointer Coordinate (horizontal)
**DX** = Pointer Coordinate (vertical)
**SI** = Last raw vertical mickey count read from mouse
**DI** = Last raw horizontal mickey count read from mouse.

**Note:** Because the DS register contains the mouse driver data segment, the user's subroutine must set it to its own data segment value.

Because this call occurs at interrupt time, it should process the information quickly and return. If it does not, interrupts could be lost.

## INT 33H-21 Query Save Mouse State Storage Requirements

**Purpose** Get the size of the buffer required to store the current state of the mouse driver.

**Input Parameters:**

AX = Function code of 21
BX = Not used
CX = Not used
DX = Not used

**Return codes/data values:**

AX = Unchanged
BX = Size of buffer required to store the mouse state
CX = Unchanged

DX = Unchanged

## INT 33H-22 Save Mouse Driver State

**Purpose** Save the mouse driver state in a user buffer.

This function moves the mouse driver data, required to restore the mouse driver state, into the user defined buffer. This function is used in conjunction with function 23 when the mouse driver state must be saved and later restored.

**Input Parameters:**

AX = Function code of 22
BX = Not used
CX = Not used
DX = Offset of buffer
ES = Segment of buffer

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

## INT 33H-23 Restore Mouse Driver State

**Purpose** Restore the mouse driver state from a user buffer.

This function restores mouse driver data previously saved by function 22 (Save Mouse Driver State). This function is used in conjunction with function 23 when the mouse driver state must be saved and later restored.

**Input Parameters:**

AX = Function code of 23
BX = Not used
CX = Not used
DX = Offset of buffer
ES = Segment of buffer

**Return codes/data values:**

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

# CLOCK$ Device Driver

OS/2 assumes that the CMOS real-time clock is available in the system. The CLOCK$ device defines and performs functions like any other character device except that it is identified by a bit in the attribute word. OS/2 uses this bit to identify the device driver, and therefore, this device can take any name. The device has been named "CLOCK$" to avoid possible conflicts with any files named "CLOCK."

OS/2 on the IBM Personal Computer AT makes use of the clock/calendar chip for its clock ticks. This device is not available on other models of the PC family and is therefore not programmed by the DOS mode applications. When the DOS mode is in the foreground, the regular 18.2 HZ clock ticks arrive and are intercepted and/or disposed of in a DOS 3.3 compatible manner. When the DOS mode is in the background, the 18.2 HZ clock is masked off. The clock/calendar clock continues to run in both modes.

The CLOCK$ device itself -- the driver that sets and returns time of day -- is dual mode and services both modes. There is no reservation of the device; the time can be set from either mode.

The CLOCK device is unique because OS/2 reads or writes a 6-byte sequence which encodes the date and time. Writing to this device sets the date and time, and reading from it gets the date and time.

The following figure illustrates the binary time format used by the CLOCK device:

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 |
|---|---|---|---|---|---|
| Days since 1-1-80<br>Low-byte Hi-byte | | Minutes | Hours | Sec/100 | Seconds |

Figure 6-1. CLOCK device time format

The CLOCK$ device driver sets and maintains the following fields in the Global InfoSeg:

TIME

- Time from 1-1-1970 in seconds
- Milliseconds
- Hours
- Minutes
- Seconds
- Hundredths
- Timer interval

DATE

- Day
- Month
- Year
- Day of week

The CLOCK$ device driver ensures that the date, time from 1-1-70 in seconds, and time of day (hours, minutes, seconds) fields remain synchronized with the CMOS clock and that the hundredths of seconds field is correctly synchronized with the seconds field.

# Console Device Drivers (Screen and Keyboard)

In OS/2, the generic console device driver has been replaced by two independent device drivers:

Screen or console output (SCR$)
Keyboard input (KBD$).

The KBD$ device driver supports the OS/2 interrupt-driven architecture. The SCR$ and KBD$ device drivers are part of the resident device driver set.

## Keyboard Device Driver KBD$

The keyboard device driver interfaces the physical keyboard to applications through various levels of routines.

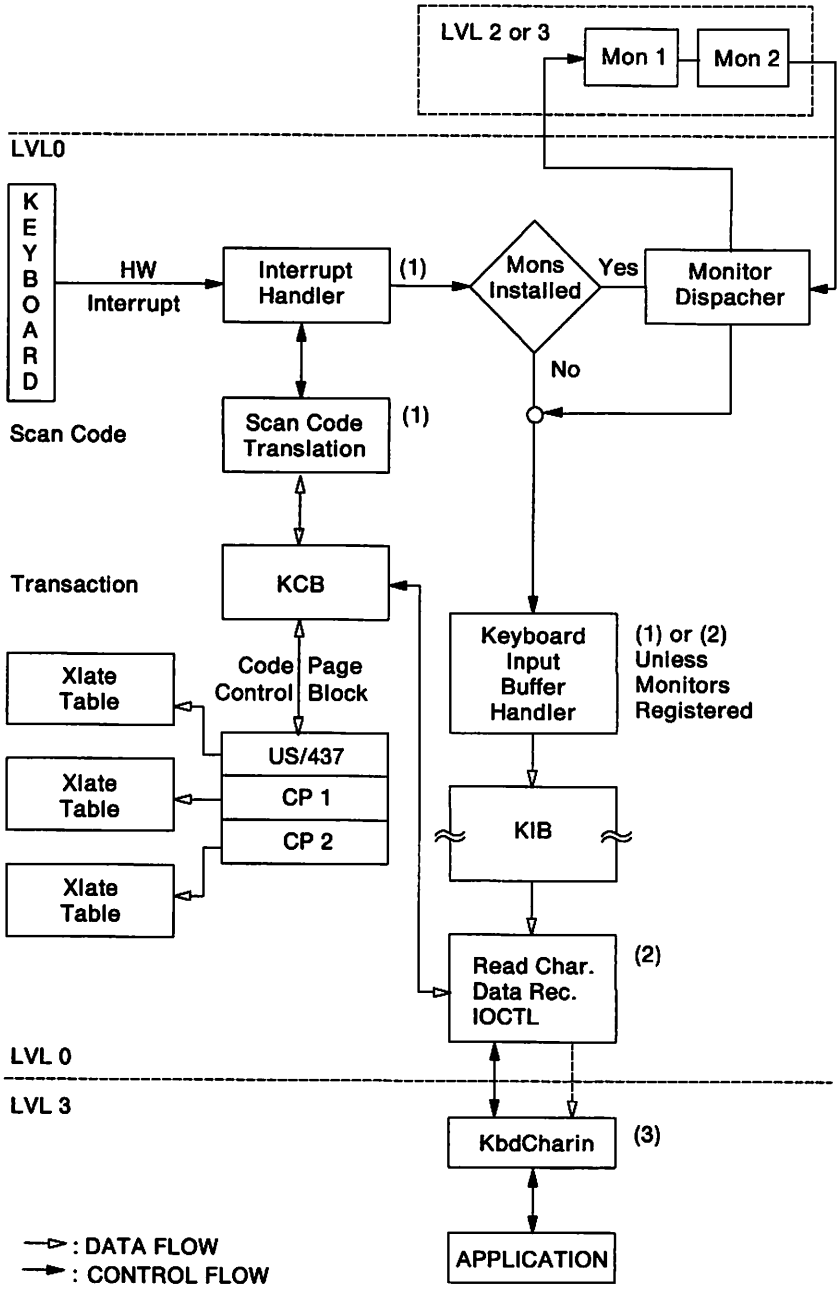



Figure 6-2. Keyboard System Structure

In Figure 6-2:

(1) Is part of the device driver interrupt handler
(2) Is part of the device driver strategy routine
(3) Is part of the base subsystem/router.

The device driver receives the make-and-break keystroke scan codes along with special Keyboard hardware codes and performs one or more of the following operations:

- Translates the scan code to an ASCII character
- Recognizes the key as a special key and signals the appropriate routine for processing
- Passes the translated scan code's character data rcord to the monitor dispatcher for further custom processing by monitors
- Places the translated scan code's character record into the appropriate session's keyboard input buffer.

The keyboard device driver supports code page switching. The following is a result of this support:

- Each handle may use one of two system-wide code pages. One is the current and the other may be swapped to. Each handle may also use the PC US 437 code page.
- The code pages to use are defined in CONFIG.SYS with the CODEPAGE and DEVINFO commands.
- Code pages specified with the CODEPAGE command in the CONFIG.SYS are for one language only. However, it is possible that code page support for different languages could result if default code pages are not accepted during execution of the KEYB command. This could occur when attempting to load translate tables in a nonsupported code page for that language.
- The user is allowed, through the KEYB command, to change the language layout of the keyboard.
- The user may control, by the KbdSetCp subsystem function or the CHCP command, which of the two code pages is used in the translation of scan codes.
- Two subsystem functions support code page switching:

  KbdSetCp - Set to an installed code page, load if necessary.

  KbdGetCp - Get the current in-use code page.
- KbdSetCustXt - Adds a custom code page option
- KbdXlate - Translates a scan code
- KbdSetCp, KbdGetCp,and KbdXlate may be replaced in the subsystem by the use of the KbdRegister function.
- Code page number, language ID of the translation table, subcountry ID, language default table indicator, and keyboard type indicator can all be found in the Translation table header of each country's keyboard layout. See Category 4, IOCtl 50H for further keyboard translation table details.

## Keyboard Initialization

At the end of CONFIG.SYS processing, after the CODEPAGE and DEVINFO statements have been processed, the code pages are then initialized. This is done by calling KbdSetCp.

## Keyboard Run Time Operation

When a session is started, a default logical keyboard will already exist. This keyboard is identified to the subsystem by a zero handle. Any program may share the keyboard by using handle zero.

If multiple programs use the default keyboard, they must coordinate their access to it. The default keyboard logically terminates when the session terminates.

If a program wants a logical keyboard separate from the default logical keyboard, it does a KbdOpen. This open creates a new logical keyboard, but does not make the physical to logical bond. As a result of an open, a handle unique within a process is returned to the caller which is later used to

identify the logical keyboard. The handle ownership is tied to the process; handles are not inherited. Use of KbdOpen does not prohibit the process from using the default keyboard.

To make the physical to logical bond, the process issues the KbdGetFocus, using the handle identifying the logical keyboard. Once the bond is made, the logical keyboard may receive keystrokes. Note that type-ahead keystrokes are not possible before the bond is made. The Keyboard API may only be used when the bond is made or with handle 0 when no other handle has the bond.

The bond represents a foreground keyboard; one exists for each session. This is either the default or a created logical keyboard.

Breaking the bond is done with the KbdFreeFocus call. If other threads have a KbdGetFocus outstanding, the thread having the highest priority will get the bond. If there are no KbdGetFocus calls outstanding, the physical keyboard will revert to the default keyboard.

A logical keyboard is destroyed with a KbdClose. The close will do a free focus, flush buffer, and de-allocate the KCB and related memory. Close will be done by the process kill mechanism, if not done by the program.

## Keystroke Monitors

Some applications need to view the raw keystrokes as they arrive from the keyboard at interrupt time. These applications may wish to consume, modify, or replace keystrokes. This is made possible by the keystroke monitor function.

When the DosMonReg call is used with keyboard devices, the *INDEX* parameter indicates the OS/2 session, from 0 to 15 (see DosGetInfoSeg). -1 indicates the session of the calling thread.

The size of the keyboard device driver's monitor chain buffer is 16 bytes. This is the value to be used in calculating the sizes of the input/output buffers required for the DosMonReg call.

The keyboard device driver supports device monitors. The keyboard device driver passes its information to the monitors in packets. Figure 6-3 describes the content and format of keystroke monitor packets.

| MonFlagWord: | | Word |
|---|---|---|
| CharData Record | XlatedChar: | Byte |
| | XlatedScan: | Byte |
| | DBCS Status: | Byte |
| | DBCS Shift | Byte |
| | Shift State: | Word |
| | Milliseconds: | DWord |
| KbdDDFlagWord | | Word |

Figure 6-3. Keystroke Monitor Data Packet Definition

**MonFlagWord Lower Byte:** Monitor Dispatcher Flags

***Bit#*** ***Meaning***

7-3 RESERVED = 0
Should be passed untouched on packets being passed on. Should be set to 0 on packets that are being inserted by a monitor.

2 FLUSH
This is a flush packet. No other information in the packet has meaning. Monitor should flush its internal buffers and pass the packet quickly.

1 CLOSE
Not used by keystroke monitors.

0 OPEN
Not used by keystroke monitors.

**MonFlagWord Upper Byte:** Original Scan code, as read from the hardware. If 0, this packet was inserted for other reasons, see "KbdDDFlagWord." Monitors pass this field untouched. Monitors should put a 0 here if they insert a packet.

**CharData Record:** Same as defined by the KbdCharIn function call in the *OS/2 Version 1.2 Programming Reference*.

**KbdDDFlagWord**

***Bit#*** ***Meaning***

15,14 Available.
These bits are available for communication between monitors. They are not used by the device driver in any way. The monitor applications coordinate the use of these flags.

13-10 RESERVED = 0
Monitors should pass these flags as is. They should set these flags to 0 in packets that they create.

9 ACCENTED
This key was translated using the previous key passed, which was an accent key (Refer to 10H ACCENT KEY on page 6-69). In the case where an accent key is pressed and the following key doesn't use the accent, a packet containing the accent character itself is first passed, with this bit set (and the scan code field of MonFlagWord, see above, would be 0, indicating a non-key-generated record). Then a valid packet containing that following keystroke is passed, without this bit set.

8 MULTIMAKE
The translation process sees this scan code as a typematic repeat of a toggle key or a shift key. Because toggle and shift keys only change state on the first make after each key-break, no state information is changed (for example, the NUMLOCK toggle bit in the shift status word is not changed, even though this may be the NumLock key). If this key is a valid character, it will not go into the KIB once this bit is set.

7 SECONDARY
The scan code prior to the one in this packet was the SECONDARY KEY PREFIX (see below).

6 KEY BREAK
This record is generated by the release (the BREAK) of the key involved.

5-0 Key Type

Numeric field that tells the device driver that this is a key that requires action. The number in this field is filled in during the translation of the scan-code. The value here allows the device driver to act on keystrokes without regard for what scan codes the keyboard uses or character codes that the current translation process may be using. The following values are currently defined:

**VALUE FOR KEYS THAT ARE ALWAYS PLACED IN THE KIB**

0 - No special action. Always place in KIB.

**VALUES ACTED ON PRIOR TO PASSING PACKET TO MONITORS**

Except for the final keystroke of the DUMP key sequences, all of these values are passed on to the monitors. They will NOT be placed in the KIB. The XlatedChar and XlatedScan fields are undefined for these values:

***Key Type Meaning***

01H ACK
This scan code was a keyboard acknowledge. IBM Personal Computer AT attached keyboards would set this value on a FAH scan code.

02H SECONDARY KEY PREFIX
This scan code was a prefix scan code generated by the Enhanced Keyboard, indicating that the next scan code coming is one of the secondary keys that exists on that keyboard. Usually set on a E0H scan code or a hex E1 scan code.

03H KBD OVERRUN
This scan code was an over-run indication from the keyboard. On an IBM Personal Computer AT attached keyboard, this value would be set on a FFH scan code.

04H RESEND
This scan code was a "resend" request from the keyboard. On an IBM Personal Computer AT attached keyboard this value would be set on a FEH scan code.

05H REBOOT KEY
This scan code is the key that completes the multi-key restart sequence. On an IBM Personal Computer AT attached keyboard this value would be used when the Ctrl + Alt + Delete sequence is seen.

06H DUMP KEY
This scan code is is the key that completes the multi-key Stand Alone Dump request sequence. On an IBM Personal Computer AT attached keyboard, this value would be used on completion of the second consecutive press of Ctrl + Alt + NumLock without other keystrokes between the two presses.

07H-0AH See -- Heading 'VAOAPP' unknown --.

0BH Invalid Accent Combination
This scan code is one that follows an accent scan code, but the combination was not valid and neither key is put in the KIB.

**Note:** This is set if the Canadian-French code pages are in use.

0C-0FH RESERVED = 0
(Will be treated as UNDEFINED, see entry 3FH.)

**VALUES ACTED ON AFTER PASSING PACKET TO MONITORS**

Except where noted, these will be placed in the KIB when the device driver is in BINARY mode, but will not be placed in the KIB when the device driver is in ASCII mode. Also noted are those that never get placed in the KIB.

***Key Type Meaning***

07H SHIFT KEY
This scan code translated as a shift key and has affected the shift status fields of the CharData record but does not generate a defined character, so it will not be placed in the KIB. The XlatedChar field is undefined. The scan code field will be 0.

08H PAUSE KEY
This scan code was translated as the key sequence meaning PAUSE. On an IBM Personal Computer AT attached keyboard, this value will be used when the Ctrl + NumLock sequence is seen. The key itself will not be placed in the KIB.

09H PSEUDO-PAUSE KEY
This scan code is translated into the value that is treated as the PAUSE key when the device driver is in ASCII mode. On most keyboards this would be when the Ctrl + S combination is seen. The key itself will not be placed in the KIB.

0AH WAKE-UP KEY
This scan code is the key that follows a PAUSE KEY or PSEUDO-PAUSE KEY which will cause the pause state to be ended. The key itself will be not be placed in the KIB.

10H ACCENT KEY
This scan code was translated as a key to be used in translating the NEXT key to come in. The packet containing this value is passed when the accent key is hit, but it is not put into the KIB (unless the ACCENTED bit is on). Refer to ACCENTED bit on page 6-67. The next key determines this decision. If the next key is one that can be accented, then it will be passed by itself, with the ACCENTED bit on. If that next key cannot be accented by this accent, then two packets are passed. The first contains the character to print for the accent itself, has this value (ACCENT KEY) and has the ACCENTED flag (which says it's okay to put it in the KIB). The second packet contains a regular translation of that following key.

**Note:** The two packets get passed for every language except Canadian-French. See entry 0BH.

11H BREAK KEY
This scan code was translated as the key sequence meaning BREAK. On the IBM Personal Computer AT attached keyboard, this value will be used where the Ctrl + Break sequence is seen.

12H PSEUDO-BREAK KEY
This scan code is translated into the value that is treated as the BREAK key when the device driver is in ASCII mode. On most keyboards this would be when the Ctrl + C combination is seen. Note that the event generated by this key is separate from that generated by the BREAK KEY when in the binary mode.

13H PRINT SCREEN KEY
This scan code was translated as the key sequence meaning PRINT SCREEN. On an IBM Personal Computer AT attached keyboard, this value will be used where the Shift-PrtSc sequence is seen.

14H PRINT ECHO KEY
This scan code was translated as the key sequence meaning PRINT ECHO. This value will be used where the Ctrl + PrtSc sequence is seen.

15H PSEUDO-PRINT ECHO KEY
This scan code is translated into the value that is treated as the PRINT ECHO key when the device driver is in ASCII mode. On most keyboards this would show as the Ctrl + P combination.

16H PRINT-FLUSH KEY
This scan code is translated into the key sequence Print-Flush. This value will be used where the Ctrl + Alt + PrtSc sequence is seen.

17-2FH RESERVED = 0
Will be treated as undefined, see entry 3FH.

**VALUES FOR PACKETS NOT GENERATED BY A KEYSTROKE**

***Key Type Meaning***

30-37H RESERVED

38-3EH RESERVED
Will be treated as UNDEFINED, see entry 3FH.

**VALUE FOR KEYS THAT THE TRANSLATION PROCESS DOES NOT RECOGNIZE**

***Key Type Meaning***

3FH UNDEFINED
This scan code, or its combination with the current shift state, was not recognized in the translation process.

## Special Key Processing

OS/2 examines each incoming keyboard character to determine if it should cause an asynchronous signal to be sent to some process (for example, Ctrl+C). The keyboard device driver responds to the keystrokes listed in Table 6-4.

Table 6-4. Special Key Processing

| Keys | Mode | Event Signalled or Signal Handler Called | Value Placed in KIB |
|---|---|---|---|
| Ctrl+C | Binary | Nothing | 03H |
| Ctrl+C | ASCII | SIGINTR | Nothing |
| Ctrl+S | Binary | Nothing | 13H |
| Ctrl+S | ASCII | event_CtrlScrLk | Nothing |
| Ctrl+P | Binary | Nothing | 10H |
| Ctrl+P | ASCII | event_CtrlPrtSc | Nothing |
| Ctrl+Break | Binary | SIGBREAK | 00:00H |
| Ctrl+Break | ASCII | SIGINTR | Nothing |
| Ctrl+NumLock | Binary | event_CtrlScrLk | Nothing |
| Ctrl+NumLock | ASCII | event_CtrlScrLk | Nothing |
| Ctrl+PrtSc | Binary | event_CtrlPrtSc | Nothing |
| Ctrl+PrtSc | ASCII | event_CtrlPrtSc | Nothing |
| PrtSc | Binary | event_ShftPrtSc | Nothing |
| PrtSc | ASCII | event_ShftPrtSc | Nothing |

**Notes:**

1. The value xxH represents the translated ASCII field in the character data record.
2. The value xx:yyH represents the translated ASCII:translated scan code fields of the character data record.
3. Ctrl+NumLock has no effect on an IBM Enhanced keyboard; the Pause key provides this function.

| *Key* | *Meaning* |
|---|---|
| Hot Key | Change sessions |
| Ctrl+Alt+Del | System IPL (Restart) |
| Ctrl+Alt+NumLock | Pressed twice means system dump |
| PrtScr | Print the current display screen (Shift+PrtScr on an IBM Personal Computer AT keyboard and Print Screen on an IBM Enhanced keyboard) |

In addition to passing individual characters, the driver must specifically recognize the character (or character sequence, or other special action) that indicates the special signal or hot key to the session manager. When this event is recognized, the SendEvent device helper function is invoked.

Note that the system hot key is defined by IOCtl Category 4 Function 56H (Set Session Manager Hot Key).

## Compatibility Operations

The DOS mode runs in its own session; the session mechanism keeps OS/2 mode programs from affecting the compatibility screen. When the DOS mode is no longer in the foreground, the old application is frozen, keeping it from affecting the OS/2 screen. The DOS mode has, in effect, an independent SCRN and KBD device driver in the form of a DOS mode CON driver. This driver is compatible with the ROM INT10 and INT16 entries and their associated low-memory data structures.

# EGA.SYS and INT 2F Screen Switch Notification

For some DOS mode EGA applications, OS/2 will not be able to switch from DOS mode to OS/2 mode and then back again. Upon return to the DOS mode application, the screen will be incorrect. The DOS mode EGA applications that do not run successfully are:

- Applications that download fonts into a character generator block other than block 0. Character generator block 0 is supported. (See EGA ROM BIOS function call INT 10H, AH=11H.)
- Graphic mode applications that use more than one display page.
- Advanced graphics mode applications that write directly to the registers on the EGA adapter.

To supplement OS/2 screen switching support, a DOS mode application can be written to use the EGA Register Interface. Alternatively, a DOS mode application can be notified on a screen switch through Multiplex Interrupt 2FH, AH=40H. These two mechanisms are described in the following sections.

**Note:** On an IBM Personal System/2 or an IBM Personal Computer AT with the IBM Personal System/2 Display Adapter, the registers on the adaptor are both readable and writable. For these configurations, OS/2 reads and saves the registers on a screen switch away from DOS mode and restores the registers upon return to DOS mode.

For configurations including the IBM Personal System/2 Display Adapter 8514/A, when the 8514/A display adapter is in an advanced function mode, OS/2 does not save the physical display buffer when switching away from DOS mode. Therefore, end users should be cautioned to complete any 8514/A advanced function mode application before switching to OS/2 mode.

## EGA.SYS Device Driver

EGA.SYS is a device driver which provides support for the EGA Register Interface in the DOS mode. In the DOS mode, to support advanced graphics modes D, E, F and 10, the mouse pointer draw device driver must save/restore the EGA registers. Because the EGA registers are not readable, this can be done only if the application cooperates in setting the registers in the first place. Rather than doing I/O directly to the registers on the adapter, the application sets the registers through the EGA Register Interface.

### EGA Register Interface

The EGA Register Interface is a library of ten functions supported for DOS mode, advanced graphics (modes D, E, F, and 10) applications. These functions do the following:

- Read from or write to one or more of the EGA write-only registers.
- Define default values for the EGA write-only registers, reset the EGA registers to these default values, or return the default values.
- Check whether the EGA Register Interface is present and, if so, return its version number.

When your application uses the EGA Register Interface, OS/2 maintains a backup copy of (shadows) how the EGA registers are set. Then, if the operator switches away from and later returns to your application, the registers will be restored properly.

It is not necessary to use the EGA Register Interface to set the mode, color palette, or palette registers. Instead, use ROM BIOS function call INT 10H with AH = 00H, 0BH, or 10H, respectively.

**How to Call the EGA Register Interface:** To call EGA Register Interface functions from an assembly language program:

1. Load the registers with the required parameter values.
2. Execute software interrupt 10H.

Values returned by the EGA Register Interface functions are placed in registers.

**EGA Register Interface Restrictions: Functions not supported**

Multiple display pages in graphics modes are not supported. Fonts may be loaded (using ROM BIOS INT 10H with AH = 11H) into character generator block 0 only.

**Attribute Controller Registers**

Before your application program uses the Attribute Controller registers (I/O address 3C0H) in an extended interrupt 10H call, it must set the flip-flop that selects the address or data register so that it selects the address register (by doing an input from I/O port 3BAH or 3DAH). The flip-flop is always reset to this state upon return from the extended interrupt 10H call.

Interrupt routines that access the attribute chip must also leave the flip-flop set to the address register upon return from the interrupt. Note, if your application program sets the flip-flop so that it selects the Data register and expects the flip-flop to remain in this state, your application must disable interrupts between the time it sets the flip-flop to the Data register state and the last time the flip-flop is assumed to be in this state.

**Sequencer Memory Mode Register**

When the Sequencer Memory Mode register (I/O address 3C5H, data register 4) is accessed, the sequencer produces a glitch on the CAS lines that may cause problems with video random access memory. As a result, your application program cannot use the EGA Register Interface to read from or write to this register. Instead, use the following procedure to safely alter this register:

1. Disable interrupts.
2. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 0.
3. Read/modify/write the Sequencer Memory Mode register.
4. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 1.
5. Enable interrupts.

**Input Status Registers**

Your application program cannot use the EGA Register Interface to read Input Status registers 0 (I/O address 3C2H) and 1 (I/O address 3BAH or 3DAH). If your program must read these registers, it should do so directly.

**Graphics Controller Miscellaneous Register**

When the Graphics Controller Miscellaneous register (I/O address 3CFH, data register 6) is accessed, a glitch on the CAS lines occurs that may cause problems with video random access memory. As a result, your application program should not use the EGA Register Interface to read from or write to this register.

EGA Register Interface function F6 does not alter the state of the Graphics Controller Miscellaneous register. Instead, use the following procedure to safely alter this register:

1. Disable interrupts.
2. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 0.
3. Read/modify/write the Graphics Controller Miscellaneous register.
4. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 1.
5. Enable interrupts.

**EGA Register Interface Functions:** This section describes each EGA Register Interface function in detail. The following list shows these functions by function number:

| ***Number (Hex)*** | ***Function*** |
|---|---|
| F0 | Read one register |
| F1 | Write one register |
| F2 | Read register range |

F3 Write register range
F4 Read register set
F5 Write register set
F6 Revert to default registers
F7 Define default register table
F8 Read default register table
FA Interrogate driver

**Note:** Calls F9H, and FBH through FFH are reserved.

Each function description includes:

- The parameters required to make the call (input) and the expected return values (output).
- Any special considerations regarding the function.

If the function description does not specify an input for a parameter, you do not need to supply a value for that parameter before making the call. If the function description does not specify an output value for a parameter, the parameter's value is the same before and after the call.

**Note:** The EGA Register Interface does not check input values, so be sure that the values you load into the registers before making a call are correct.

## Function F0 - Read One Register

**Purpose** Function F0 reads data from a specified register on the EGA.

**Input:**

AH = F0H

BX = Pointer for pointer/data chips:

BH = 0
BL = Pointer

Ignored for single registers

DX = Port number:
Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

**Output:**

AX: Restored
BH: Restored
BL: Data
DX: Restored

All other registers restored.

**Example:** The following example saves the contents of the Sequencer Map Mask register in "myvalue:"

```
myvalue db   ?
        mov  ah, 0f0h       ; f0 = read one register
        mov  bx, 0002h      ; bh = 0 / bl = map mask
                            ; index
        mov  dx, 0008h      ; dx = sequencer
        int  10h            ; get it!
        mov  myvalue, bl    ; save it!
```

The following example saves the contents of the Miscellaneous Output register in "myvalue:"

```
myvalue db   ?
        mov  ah, 0f0h        ; f0 = read one register
        mov  dx, 0020h       ; dx = miscellaneous
                             ;  output register
        int  10h             ; get it!
        mov  myvalue, bl     ; save it!
```

## Function F1 - Write One Register

**Purpose** Function F1 writes data to a specified register on the EGA.

When your application program returns from a call to function F1, the contents of registers BH and DX are not restored. Your program must save and restore these registers if desired.

**Input:**

AH = F1h

BL = Pointer for pointer/data chips
Data for single registers,

BH = Data for pointer/data chips
(ignored for single registers)

DX = Port number:
Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

**Output:**

AX: Restored
BL: Restored
BH: Not restored
DX: Not restored

All other registers restored.

**Example:** The following example writes the contents of "myvalue" into the CRT Controller Cursor Start register:

```
myvalue db   3h
        mov  ah, 0f1h        ; f1 = write one register
        mov  bh, myvalue     ; bh = data from myvalue
        mov  bl, 000ah       ; bl = cursor start index
        mov  dx, 0000h       ; dx = crt controller
        int  10h             ; write it!
```

The following example writes the contents of "myvalue" into the Feature Control register:

```
myvalue db   2h
        mov  ah, 0f1h        ; f1 = write one register
        mov  bl, myvalue     ; bl = data from myvalue
        mov  dx, 0028h       ; dx = feature control register
        int  10h             ; write it!
```

## Function F2 - Read Register Range

**Purpose** Function F2 reads data from a specified range of registers on the EGA. A range of registers is defined to be several registers on a single chip that have consecutive indexes. This call is applicable for pointer/data chips.

**Input:**

AH = F2h

CH = Starting pointer value

CL = Number of registers (must be > 1)

DX = Port number:

0h: CRT Controller (3?4h)
8h: Sequencer (3C4h)
10h: Graphics Controller (3CEh)
18h: Attribute Controller (3C0h)

? = B for monochrome modes or D for color modes

ES:BX = Points to table of one-byte entries (length = value in CL). On return, each entry is set to the contents of the corresponding register.

**Output:**

AX: Restored
BX: Restored
CX: Not restored
DX: Restored
ES: Restored

All other registers restored.

**Example:** The following example saves the contents of the Attribute Controller Palette registers in "paltable:"

```
paltable db  16 dup (?)
        mov  ax, ds              ; assume paltable in
                                 ;  data segment
        mov  es, ax              ; es = data segment
        mov  bx, offset paltable ; es:bx = paltable
                                 ;  address
        mov  ah, 0f2h            ; f2 = read register
                                 ;  range
        mov  cx, 0010h           ; ch = start index
                                 ;  of 0
                                 ; cl = 16 registers
                                 ;  to read
        mov  dx, 0018h           ; dx = attribute
                                 ;  controller
        int  10h                 ; read them!
```

## Function F3 - Write Register Range

**Purpose** Function F3 writes data to a specified range of registers on the EGA. A range of registers is defined to be several registers on a single chip that have consecutive indexes. This call is applicable for the pointer/data chips.

**Input:**

AH = F3h

CH = Starting pointer value

CL = Number of registers (must be > 1)

DX = Port number:

0h: CRT Controller (3?4h)
8h: Sequencer (3C4h)
10h: Graphics Controller (3CEh)
18h: Attribute Controller (3C0h)

? = B for monochrome modes or D for color modes

ES:BX = Points to table of one-byte entries (length = value in CL). Each entry contains the value to be written to the corresponding register.

**Output:**

AX: Restored
BX: Not restored
CX: Not restored
DX: Not restored
ES: Restored

All other registers restored

**Example:** The following example writes the contents of "cursloc" into the CRT Controller Cursor Location High and Cursor Location Low registers.

```
cursloc db   01h, 00h            ; cursor at page
                                 ;  offset 0100h
        mov  ax, ds              ; assume cursloc in
                                 ;  data segment
        mov  es, ax              ; es = data segment
        mov  bx, offset cursloc  ; es:bx = cursloc
                                 ;  address
        mov  ah, 0f3h            ; f3 = write register
                                 ;  range
        mov  cx, 0e02h           ; ch = start index
                                 ;  of 14
                                 ; cl = 2 registers to
                                 ;  write
        mov  dx, 0000h           ; dx = crt controller
        int  10h                 ; write them!
```

## Function F4 - Read Register Set

**Purpose** Function F4 reads data from a set of registers on the EGA. A set of registers is defined to be several registers that may or may not have consecutive indexes, and that may or may not be on the same chip.

**Input:**

AH = F4H

CX = Number of registers (must be > 1)

ES:BX = Points to table of records with each entry in this format:

Bytes 1-2: Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

Byte 3: Pointer value (0 for single registers)

Byte 4: EGA Register Interface fills in data read from register specified in bytes 1-3.

**Output:**

AX: Restored
BX: Restored
CX: Not restored
ES: Restored

All other registers restored.

**Example:** The following example saves the contents of the Miscellaneous Output register, Sequencer Memory Mode register, and CRT Controller Mode Control register in "results:"

```
outvals dw   0020h        ; miscellaneous output
                          ;  register
        db   0            ; 0 for single registers
        db   ?            ; returned value
        dw   0008h        ; sequencer
        db   04h          ; memory mode register
                          ;  index
        db   ?            ; returned value
        dw   0000h        ; crt controller
        db   17h          ; mode control register
                          ;  index
        db   ?            ; returned value

results db   3 dup (?)
        mov  ax, ds       ; assume outvals in
                          ;  data segment
        mov  es, ax       ; es = data segment
        mov  bx, offset outvals  ; es:bx = outvals
                                 ;  address
        mov  ah, 0f4h     ; f4 = read register set
        mov  cx, 3        ; number of entries in
                          ;  outvals
        int  10h          ; get values into outvals
        mov  si, 3              ; move the returned
                                ;  values from
        add  si, offset outvals  ;  outvals
        mov  di, offset results  ;  to results
        mov  cx, 3        ; 3 values to move

loop:   mov  al, [si]     ; move one value from
        mov  [di], al     ;  outvals to results
        add  si, 4        ; skip to next source byte
        inc  di           ; point to next destination
                          ;  byte
        loop loop
```

## Function F5 - Write Register Set

**Purpose** Function F5 writes data to a set of registers on the EGA. A set of registers is defined to be several registers that may or may not have consecutive indexes, and that may or may not be on the same chip.

**Input:**

AH = F5h

CX = Number of registers (must be > 1)

ES:BX = Points to table of values with each entry in this format:

Bytes 1-2: Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

Byte 3: Pointer value (0 for single registers)

Byte 4: Data to be written to register specified in bytes 1-3

**Output:**

AX: Restored

BX: Restored
CX: Not restored
ES: Restored

All other registers restored.

**Example:** The following example writes the contents of "outvals" to the Miscellaneous Output register, Sequencer Memory Mode register, and CRT Controller Mode Control register:

```
outvals dw   0020h          ; miscellaneous output
                            ;  register
        db   0              ; 0 for single registers
        db   0a7h           ; output value
        dw   0008h          ; sequencer
        db   04h            ; memory mode register index
        db   03h            ; output value
        dw   0000h          ; crt controller
        db   17h            ; mode control register index
        db   0a3h           ; output value
        mov  ax, ds         ; assume outvals in
                            ;  data segment
        mov  es, ax         ; es = data segment
        mov  bx, offset outvals  ; es:bx = outvals
                                 ;  address
        mov  ah, 0f5h       ; f5 = write register set
        mov  cx, 3          ; number of entries in
                            ;  outvals
        int  10h            ; write the registers!
```

## Function F6 - Revert to Default Registers

**Purpose** Function F6 restores the default settings of any registers that your application program has changed through the EGA Register Interface. The default settings are defined in a call to function F7 (described in the next section).

**Input:**

AH = F6H

**Output:**

All registers restored.

**Example:** The following example restores the default settings of the EGA registers:

```
mov  ah, 0f6h     ; f6 = revert to default
                  ;  registers
int  10h          ; do it now!
```

## Function F7 - Define Default Register Table

**Purpose** Function F7 defines a table containing default values for any pointer/data chip or single register. If you define default values for a pointer/data chip, you must define them for all registers within that chip.

**Input:**

AH = F7h

DX = Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

ES:BX = Points to table of one-byte entries. Each entry contains the default value for the corresponding register. The table must contain entries for all registers.

**Output:**

AX: Restored
BX: Not restored
DX: Not restored
ES: Restored

All other registers restored.

**Example:** The following example defines default values for the Attribute Controller:

```
attrdflt db  00h, 01h, 02h, 03h, 04h, 05h, 06h, 07h
         db  10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h
         db  08h, 00h, 0fh, 00h
         mov  ax, ds              ; assume attrdflt in
                                  ;  data segment
         mov  es, ax              ; es = data segment
         mov  bx, offset attrdflt ; es:bx = attrdflt
                                  ;  address
         mov  ah, 0f7h            ; f7 = define default
                                  ;  register table
         mov  dx, 0018h           ; dx = attribute
                                  ;  controller
         int  10h                 ; do it!
```

The following example defines a default value for the Feature Control register:

```
featdflt db  00h
         mov  ax, ds              ; assume featdflt in
                                  ;  data segment
         mov  es, ax              ; es = data segment
         mov  bx, offset featdflt ; es:bx = featdflt
                                  ;  address
         mov  ah, 0f7h            ; f7 = define default
                                  ;  register table
         mov  dx, 0028h           ; dx = feature
                                  ;  control register
         int  10h                 ; do it!
```

## Function F8 - Read Default Register Table

**Purpose** Function F8 reads the table containing default register values for any pointer/data chip or single register.

**Input:**

AH = 0F8H

DX = Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

ES:BX = Points to a table into which the default values are returned. The table must have room for the full set of values for the pointer/data chip or single register specified.

**Output:**

AX: Restored
BX: Not restored
DX: Not restored

ES: Restored

All other registers restored.

## Function FA - Interrogate Driver

**Purpose** Function FA returns a value specifying whether the EGA.SYS driver is present.

**Input:**

AH = FAh

BX = 0

**Output:**

AX: Restored
BX: 0, if EGA.SYS driver is not present
ES:BX: Pointer to EGA Register Interface version number, if present:
Byte 1: Major release number
Byte 2: Minor release number (in 1/100ths)

**Example:** The following example interrogates the driver and displays the results:

```
gotmsg  db   "EGA.SYS driver found", 0dh, 0ah, 24h
nopmsg  db   "EGA.SYS driver not found", 0dh, 0ah, 24h
revmsg  db   "revision $"
crlf    db   0dh, 0ah, 24h
ten     db   10
        mov  bx, 0          ; must be 0 for this call
        mov  ah, 0fah       ; fa = interrogate driver
        int  10h            ; interrogate!
        or   bx, bx         ; bx = 0 ?
        jnz  found          ; branch if driver present
        mov  dx, offset nopmsg  ; assume nopmsg in data
                                ;  segment
        mov  ah, 09h        ; 9 = print string
        int  21h            ; output not found message
        jmp  continue       ; that's all for now
found:  mov  dx, offset gotmsg  ; assume gotmsg in data
                                ;  segment
        mov  ah, 09h            ; 9 = print string
        int  21h                ; output found message
        mov  dx, offset revmsg  ; assume revmsg in data
                                ;  segment
        mov  ah, 09h        ; 9 = print string
        int  21h            ; output "revision "
        mov  dl, es:[bx]    ; dl = major release
                            ;  number

        add  dl, "0"        ; convert to ASCII
        mov  ah, 2          ; 2 = display character
        int  21h            ; output major release
                            ;  number
        mov  dl, "."        ; dl = "."
        mov  ah, 2          ; 2 = display character
        int  21h            ; output a period
        mov  al, es:[bx+1]  ; al = minor release
                            ;  number
        xor  ah, ah         ; ah = 0
        idiv ten            ; al = 10ths, ah = 100ths
        mov  bx, ax         ; save ax in bx
        mov  dl, al         ; dl = 10ths
        add  dl, "0"        ; convert to ASCII
        mov  ah, 2          ; 2 = display character
        int  21h            ; output minor release
                            ;  10ths
        mov  dl, bh         ; dl = 100ths
        add  dl, "0"        ; convert to ASCII
        mov  ah, 2          ; 2 = display character
        int  21h            ; output minor release
                            ;  100ths
        mov  dx, offset crlf; assume crlf in data
                            ;  segment
        mov  ah, 09h        ; 9 = print string
        int  21h            ; output end of line
continue:                   ; the end
```

## INT 2F Screen Switch Notification

OS/2 issues a new Multiplex Interrupt (INT 2FH) to signal the following two events:

- Switching the DOS mode application to the background (AX = 4001H)
- Switching the DOS mode application to the foreground (AX = 4002H).

A DOS mode application that wishes to receive this signal must *hook* the Multiplex Interrupt vector. That is, when the application is started, it must save the current INT 2FH vector and set the INT 2FH vector to point to the application's own interrupt handler.

When the notification is received, the application must save all registers, perform whatever processing is required, restore all registers, and issue the IRET instruction to return to OS/2. Only the following forms of processing are supported:

- Modifying application and/or video memory
- Issuing ROM BIOS video service calls (INT 10H)
- Issuing EGA Register Interface calls (INT 10H)
- Programming the EGA video card directly.

**Note:** When an application is being switched to the background, if the application's INT 2F handler uses the EGA Register Interface to save the EGA registers, these registers are restored automatically when the application is returned to the foreground.

An application may receive notification that it is being switched to the background at any time. Code sequences that are sensitive to interruption can be protected with CLI/STI instructions. At the point the switch notification occurs, the application (other than its INT 2F handler) is frozen until it is returned to the foreground.

When an application's INT 2F handler receives notification with a value in AH other than 40H, the application must issue the JMP FAR instruction to branch to the previous INT 2FH vector.

# Using Extended Screen and Keyboard Control (ANSI.SYS, ANSICALL.DLL)

This section explains how you can issue special control character sequences to:

- Control the position of the cursor
- Erase text from the screen
- Set the display mode
- Redefine the meaning of keyboard keys.

ANSI extended screen and keyboard control sequences are supported in the DOS mode by ANSI.SYS, an installable device driver.

In the OS/2 mode, these control sequences are supported by ANSICALL.DLL, a dynamic link module.

**Note:** In this section, unless otherwise specified, ANSI refers to both ANSI.SYS and ANSICALL.DLL.

## Limitations/Restrictions

ANSI operates on a per-session basis.

OS/2 mode ANSI is affected when keys are reassigned in a code page environment. ANSI does not provide code page support for key reassignment in the DOS mode.

## Control Sequence Syntax

Each of the cursor control sequences is in the format:

ESC [ *parameters* COMMAND

as described in Table 6-5.

Table 6-5. Control Sequence

| | |
|---|---|
| **ESC** | The 1-byte ASCII code for ESC (1BH). It is not the three characters ESC. |
| [ | The character [. |
| *parameters* | The numeric values you specify for #. The # represents a numeric parameter. A numeric parameter is an integer value specified with ASCII characters. If you do not specify a parameter value, or if you specify a value of 0, the default value for the parameter is used. |
| COMMAND | An alphabetic string that represents the command. It is case specific. |

For example:

```
ESC [2;10H
```

could be created using BASIC as follows:

```
The IBM Personal Computer Basic
Version 3.00 Copyright IBM Corp. 1981, 1982, 1983, 1984
xxxxx Bytes free

Ok
open "sample" for output as 1
Ok
print #1, CHR$(27);"[2;10H";"x row 2 col 10"
Ok
close #1
Ok
```

Notice that "CHR$(27)" is ESC.

## Cursor Control Sequences

The following tables contain the cursor control sequences you can use to control cursor positioning.

| Cursor Position | Function |
|---|---|
| ESC [#;#H | Moves the cursor to the position specified by the parameters. The first parameter specifies the row number and the second parameter specifies the column number. The default value is 1. If no parameter is given, the cursor is moved to the home position. |

This example copies the file SAMPLE from the previous example, to CON, which places the cursor on row 2 column 10 of the screen:

```
type sample
```

| Cursor Up | Function |
|---|---|
| ESC [#A | Moves the cursor up one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. This sequence is ignored if the cursor is already on the top line. |

| Cursor Down | Function |
|---|---|
| ESC [#B | Moves the cursor down one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. The sequence is ignored if the cursor is already on the bottom line. |

| Cursor Forward | Function |
|---|---|
| ESC [#C | Moves the cursor forward one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored if the cursor is already in the rightmost column. |

| Cursor Backward | Function |
|---|---|
| ESC [#D | Moves the cursor back one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored if the cursor is already in the leftmost column. |

| Horizontal and Vertical Position | Function |
|---|---|
| ESC [#;#f | Moves the cursor to the position specified by the parameters. The first parameter specifies the line number and the second parameter specifies the column number. The default value is 1. If no parameter is given, the cursor is moved to the home position. |

| Cursor Position Report | Function |
|---|---|
| ESC [#;#R | The cursor sequence report reports the current cursor position through the standard input device. The first parameter specifies the current line and the second parameter specifies the current column. |

| Device Status Report | Function |
|---|---|
| ESC [6n | The console driver gives a cursor position report sequence on receipt of device status report. |

**Note:** Do not use the Device Status Report as part of a prompt.

This example tells ANSI to put the current cursor position (row and column) in STDIN. Then the program reads it from STDIN and outputs to STDOUT.

```
PROGRAM dsr(INPUT,OUTPUT);

  VAR
    f:FILE OF CHAR;
    key:CHAR;

  FUNCTION inkey:CHAR;                { read character  }
    VAR                               {  from the       }
      ch:CHAR;                        { keyboard buffer }
    BEGIN
      READ(f,ch);
      inkey:=ch
    END;

  BEGIN
    ASSIGN(f,'user');
    RESET(f);
        WRITE(CHR(27),'[6n');         { issue a DSR   }
        key:=inkey;                   { read up to    }
        key:=inkey;                   { first digit   }
        key:=inkey;                   { of the row    }
        WRITE('row ',inkey,inkey,'  column ');
        key:=inkey;                   { skip to column}
        WRITE(inkey,inkey)            { write column  }
  END.
```

| Save Cursor Position | Function |
|---|---|
| ESC [s | The current cursor position is saved. This cursor position can be restored with the restore cursor position sequence (see below). |

## Restore Cursor Position

| Restore Cursor Position | Function |
|---|---|
| ESC [u | Restores the cursor to the value it had when the console driver received the save cursor position sequence. |

# Erasing

The following tables contain the control sequences you can use to erase text from the screen.

| Erase In Display | Function |
|---|---|
| ESC [2J | Erases all of the screen and the cursor goes to the home position. |

| Erase In Line | Function |
|---|---|
| ESC [K | Erases from the cursor to the end of the line and includes the cursor position. |

## Controlling Display Mode

The following tables contain the control sequences you can use to set the mode of operation. They are:

- Set Graphics Rendition (SGR)
- Set Mode (SM)
- Reset Mode (RM)

| Set Graphics Rendition (SGR) | Function |
|---|---|
| ESC [#;...;#m | Sets the character attribute specified by the parameters. All following characters have the attribute according to the parameters until the next occurrence of SGR. |

| Parameter | Meaning |
|---|---|
| 0 | All attributes off (normal white on black) |
| 1 | Bold on (high intensity) |
| 4 | Underscore on (mono-compatible modes) |
| 5 | Blink on |
| 7 | Reverse video on |
| 8 | Canceled on (invisible) |
| 30 | Black foreground |
| 31 | Red foreground |
| 32 | Green foreground |
| 33 | Yellow foreground |
| 34 | Blue foreground |
| 35 | Magenta foreground |
| 36 | Cyan foreground |
| 37 | White foreground |
| 38 | Reserved |
| 39 | Reserved |
| 40 | Black background |
| 41 | Red background |
| 42 | Green background |
| 43 | Yellow background |
| 44 | Blue background |
| 45 | Magenta background |
| 46 | Cyan background |
| 47 | White background |

| Set Mode (SM) | Function |
|---|---|
| ESC [=#h<br>or ESC [=h<br>or ESC [=0h<br>or ESC [?7h | Invokes the screen width or type specified by the parameter. |

| Parameter | Meaning |
|---|---|
| 0 | 40x25 black and white |
| 1 | 40x25 color |
| 2 | 80x25 black and white |
| 3 | 80x25 color |
| 4 | 320x200 color |
| 5 | 320x200 black and white |
| 6 | 640x200 black and white |
| 7 | Wrap at end of line. (Typing past end-of-line results in new line.) |

| Reset Mode (RM) | Function |
|---|---|
| ESC [=#l<br>or ESC [=l<br>or ESC [=0l<br>or ESC [?7l | Parameters are the same as Set Mode (SM) except that parameter 7 resets wrap at end-of-line mode (characters past end-of-line are thrown away). |

## Keyboard Key Reassignment

When the application does a KbdStringIn call, the reassigned key's ASCII code is converted to the specified string and is passed back to the calling application.

OS/2 mode ANSI is affected when keys are reassigned in a code page environment. ANSI "remembers" the code page under which a key is reassigned. The keyboard subsystem checks for reassigned keys when the application calls the KbdStringIn function. When a reassigned key is detected, the ANSI support:

- Checks to see what code page the requestor is running under.
- Looks internally to see if the key has been reassigned under that code page.
- If there is a key reassignment for that code page, gives the reassignment string.
- Otherwise, gives the original ASCII codes.

A maximum storage of 64KB may be allocated to OS/2 mode ANSI reassigned key definitions.

Table 6-6 contains the control sequences you can use to redefine the meaning of keyboard keys.

Table 6-6. Keyboard Key Reassignment

| **The control sequence is:** | **Function** |
|---|---|
| ESC [#;#;...#p<br>or ESC ["string"p<br>or ESC [#;"string";#;<br>#;"string";#p<br>or any other combination of strings and decimal numbers | The first ASCII code in the control sequence defines which code is being mapped. The remaining numbers define the sequence of ASCII codes generated when this key is intercepted. However, if the first code in the sequence is 0 (NULL) the first and second code make up an extended ASCII redefinition |

**Here are some examples:**

To execute these examples, you can either:

- Create a file that contains the following statements and then use the TYPE command to display the file that contains the statement.
- Execute the command at the OS/2 prompt.

1. Reassign the **Q** and **q** key to the **A** and **a** (and the other way as well):

   Creating a File:

```
ESC [65;81p     A becomes Q
ESC [97;113p    a becomes q
ESC [81;65p     Q becomes A
ESC [113;97p    q becomes a
```

At the OS/2 Prompt:

```
prompt $e[65;81p     A becomes Q
prompt $e[97;113p    a becomes q
prompt $e[81;65p     Q becomes A
prompt $e[113;97p    q becomes a
```

2. Reassign the F10 key to a DIR command followed by a carriage return:

   Creating a File:

   ```
   ESC [0;68;"dir";13p
   ```

   At the OS/2 Prompt:

   ```
   prompt $e[0;68;"dir";13p
   ```

   The $e is the prompt command characters for ESC. The 0;68 is the extended ASCII code for the F10 key; 13 decimal is a carriage return.

3. The following example sets the prompt to display the current directory on the top of the screen and the current drive on the current line.

   ```
   prompt $e[s$e[1;30f$e[K$p$e[u$n$g
   ```

   If the current directory is C:\FILES, and the current drive is C, this example would display:

   ```
                  C:\FILES

   C>
   ```

4. The following DOS mode assembly language program reassigns the F10 key to a DIR B: command followed by a carriage return.

```
        TITLE SETANSI.ASM - SET F10 TO STRING FOR ANSI.SYS
CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME CS:CSEG,DS:CSEG
        ORG     100H
ENTPT:  JMP     SHORT START
STRING  DB      27,'[0;68;"DIR B:";13P' ;Redefine F10 key
STRSIZ  EQU     $-STRING                ;Length of above message
HANDLE  EQU     1                       ;Pre-defined file
                                        ;Handle for standard output

START   PROC    NEAR
        MOV     BX,HANDLE               ;Standard output device
        MOV     CX,STRSIZ               ;Get size of text to be sent
        MOV     DX,OFFSET STRING        ;Pass offset of string
                                        ;To be sent
        MOV     AH,40H                  ;Function="write to device"
        INT     21H                     ;Call DOS
        RET                             ;Return to DOS
START   ENDP

CSEG    ENDS
        END     ENTPT
```

# Fixed Disk and Diskette Device Driver

The fixed disk and diskette device driver provides access to disk and diskette data and controls removable diskette media. The device driver runs in a multitasking, dual-mode environment.

The device driver receives the following commands from the kernel, processes them interacting with physical devices, and returns status:

- DOS device driver commands
- Generic category 8 and 9 IOCTIs
- Software Interrupt (INT) 13H.

DOS device driver commands are generated by the kernel in order to access data files on behalf of user issued DOS API calls. The device driver provides a primitive data access mechanism and puts no interpretation on the data requested.

The device driver's physical devices are represented to the kernel as a set of logical units following the Extended DOS Partition Architecture. The device driver presents a logical unit as a sequence of 512 byte sectors.

The kernel and the device driver maintain removable volume integrity by making sure that the correct diskette is mounted. The device driver informs the kernel of media changes through the return code. The kernel can then issue interactive prompts to request the appropriate media to be mounted.

The physical characteristics of logical units are described in BIOS Parameter Block (BPB) control blocks maintained jointly by the device driver and the kernel. The BPB contains the size and physical geometry of the logical unit as well as file system fields that are not used by the device driver.

Generic IOCTL functions access the logical unit using its cylinder, head, and sector geometry. Generic IOCtls bypass the file system control of the logical unit and are intended for system applications that set up file system environments.

The device driver also receives requests to access devices through the software INT 13H. Real mode applications issue the software INT 13H to access the entire physical device through cylinder, head, and sector, without consideration for logical units. INT 13H is supported for compatibility applications.

## Configuration

The IBM PS/2 ABIOS disk device driver supports two internal 1.44MB diskette drives or one internal and one external 5.25 inch 360KB diskette drive. Fixed disk and software disk cache support is provided for up to seven physical disk drives configured as up to seven ABIOS disk Logical IDs (LID). Up to seven units are supported in any unit-LID combination. Two shared hardware interrupt levels are supported; one for diskette and one for fixed disk LIDs. Co-residency of multiple disk adapters is supported.

The IBM Personal Computer AT disk device driver supports up to two internal and one external diskette drive. In addition the IBM Personal Computer AT disk device driver supports up to two physical disk drives.

## Performance Considerations

The device driver provides a software cache for disk devices.

All requests are queued to the physical device using the SortRequest device helper routine. Requests are sorted based and on an algorithm that minimizes seek distance.

For the IBM PS/2, independent disk LIDs and units on concurrent disk LIDs will be programmed concurrently by the device driver.

### Software Disk Cache

The device driver provides software caching for disk devices for both the AT and the PS/2. PS/2 disks that are cached by the hardware will not be cached by software.

The user enters the amount of memory for software caching by using a DISKCACHE=nnnnn command in CONFIG.SYS. The maximum cache size is 7200KB. If no DISKCACHE command is in CONFIG.SYS, caching will not be enabled.

The amount of memory specified for software caching must be sufficient for control tables and 16KB for cache memory. If the amount of memory for software caching is insufficient to cache all physical disks (insufficient for control tables and 16KB of cache memory) then the caching will not be enabled and the cache memory will be returned for general use by the system.

The amount of space required for the cache control tables is described below:

```
Total  =  628 + A + B
```

Where:

```
A  =  (DASDSize * 2) / (4096 * 8)

B  =  [(CacheSize-A-628) / (4096+18)] * 18
```

| xxx | yyy |
|---|---|
| **DASDSize** | Bytes of S/W cached DASD |
| **CacheSize** | User specified memory for software cacheing |

For example, for a PS/2 with a 70 MB and 44MB fixed disk with DISKCACHE = 256 in CONFIG.SYS:

```
A      =  (114 MB * 2) / (4096 * 8)
       =  6959

B      =  [(256 KB - 6959 - 628)/ (4096+18)] * 18
       =  60 * 18
       =  1080

Total  =  628 + A + B
       =  8667
```

The total control table requirement is 8,667 bytes of memory. For cache memory, 253,477 bytes are available.

### Interrupt 13H Support

The device driver accepts real mode application INT 13H requests and passes them on to the CBIOS for processing. The requests are synchronized with regular device driver requests. Only read functions are allowed on disk devices; write operations are rejected.

### EXTDSKDD.SYS Support

The device driver provides support for the physical devices and logical volumes declared by the EXTDSKDD.SYS installable device driver.

## DOS Extended Volume Architecture

Fixed disks can be divided into a primary partition and an extended partition that contains multiple logical block devices. The extended partition is indicated by a system ID byte of 05H in the partition table of the Master Boot (Start-up) Record. This partition cannot be started, and programs that can set startable partitions (such as OS/2 FDISKPM) do not allow the partition to be marked as able to start.

**Note:** This extended partition support can be used on any fixed disk supported by OS/2.

The extended DOS partition can be created only if a primary DOS partition already exists on a startable drive. A primary DOS partition is a partition with a system ID byte of 01H, 04H, 06H, or 07H. If the drive cannot be started, then an extended DOS partition may be created without having a primary DOS partition.

**Note:** FDISKPM refers to extended volumes as "logical drives".

The extended DOS partition starts and ends on a cylinder boundary.

The extended DOS partition contains a collection of extended volumes that are linked together by a pointer in the extended volumes' extended start-up record. An extended volume consists of an extended start-up record and one logical block device. In OS/2 Version 1.0, an extended volume cannot be larger than 32 MB due to the limitations of the FAT file system. However, in OS/2 Version 1.1 this restriction has been removed from the FAT file system. An extended volume created within the extended DOS partition can be any size from one cylinder long up through the maximum available contiguous space in the extended DOS partition. An extended volume can be larger than 32MB. All extended volumes must start and end on a cylinder boundary. An extended volume will correspond to an image of a physical disk. The extended start-up record corresponds to the master start-up record at the beginning of an actual physical disk and the logical block device corresponds to the DOS partition that is pointed to by the master start-up record.

The logical block device begins with a normal DOS start-up sector if it is a DOS logical block device (system id = 1, 4, or 6). IFS (installable file system) logical block devices (system id = 7) need not start with a normal DOS boot sector. This logical block device must start on a cylinder and head boundary and follows the extended start-up record on the physical disk. The logical block device and the extended volume both end on the same cylinder boundary.

Each extended volume contains an extended start-up record, located in the first sector of the disk location assigned to it. This extended start-up record contains the 55AAH signature ID byte. This allows programs that look at the extended (master) start-up record to be compatible. This extended start-up record also contains a partition table, which can contain only two types of entries. The start-up code is not critical, as the devices are not considered startable. The start-up code may simply report a message indicating an unstartable partition if it is executed.

The partition table portion of the extended start-up record is the same as the partition table structure in the master start-up record. This structure has four partition entries of 16 bytes each. The system ID byte must be filled in for all four entries with one of the following values:

00H No space allocated in this entry.

01H DOS partition up to 16MB

04H DOS partition with 32MB > SIZE > 16MB

05H Maps out area assigned to the next extended volume. Serves as a pointer to the next extended start-up record.

06H DOS partition size > 32MB

07H Installable file system

If the system ID byte is 0 then the values in that partition table entry will be set to 0.

If OS/2 detects any values other than 01H, 04H, 06H, or 07H it will ignore that entry and not attempt to install the logical block device. This will allow future expansion of devices in this area without problems of compatibility with earlier systems.

The partition start and end fields Cylinder, Head, Sector (C,H,S) will be filled in for any of the four partition entries in an extended start-up record that have one of the above system ID bytes. This will allow a program such as FDISKPM to determine the allocated space in the extended DOS partition, as well as allowing the device drivers to determine the physical DASD area that belongs to it. The partition start and end fields (C,H,S) for the partition entry that points to the logical block device (system ID 01H, 04H, 06H, or 07H) map out the physical boundaries of the logical block device. They are offset relative to the beginning of the extended start-up record that the entry resides in. The partition start and end fields (C,H,S) for the partition entry that points to the next extended volume (system ID 05H) map out the physical boundaries of the next extended volume. They are relative to the beginning of the entire physical disk.

The relative sector and number of sector fields will be set up differently depending on what system ID byte is used. If 01H, 04H, 06H, or 07H is in the system ID field for that extended partition entry (pointer to the logical block device), the relative sector field will be set up as an offset from (and including) the start of the extended start-up record for the associated extended volume. The number of sectors (size) field will be filled in with the size of the created logical block device area, in other

words, the number of sectors mapped out by the start and stop cylinder/track/sector fields. The size of the extended volume can be calculated by adding the relative sector field and the sector size field of the associated extended start-up record.

If the system ID byte is 05H, then the relative sector field is the offset (of the next extended volume) in sectors from the start of the entire extended DOS partition. The number of sectors field is not used in this field, and will be filled with 00H's.

This architecture allows only one logical block device to be defined for each extended start-up record. Therefore, only a maximum of two partition entries at a time is used in each extended start-up record; an entry with system ID byte of (01H, 04H, 06H, or 07H) and an entry with ID of 05H, which is the pointer to the next extended volume. Although only two entries can be used, a program installing these devices will not assume that the first two entries will be the non-zero entries.

## Installing Block Devices in the Extended Partition

To install block devices, the device drivers will first install the primary DOS partitions on all physical drives if any exist. This will insure that an existing drive letter (D:) on the 81H drive will remain the same. After these devices are installed, on the 80H drive, the drivers will look for the existence of the extended DOS partition. If one exists, then it will look at the first sector of the extended DOS partition for the first extended start-up record. If there is a valid system ID (01H, 04H, 06H, or 07H) in any of the four partition entries, the device is installed and assigned the next available drive letter. This will occur before any CONFIG.SYS device drivers are loaded so the FDISK will correctly display the drive letter when space is allocated for the drive.

The first extended start-up record (in the extended DOS partition) is a special case because it is possible there will not be a device to be installed defined in the partition table. The first device might have been created and then deleted at some time, but the first extended start-up record is needed to point to the next one, if one exists. Any other extended start-up record will always have a device to be installed.

Once a device has been installed (or the special cases above occurs), the device driver will search the other partition entries for a system ID byte of 05H, indicating that another device (extended volume) exists. If a 05H is not found, there are no more logical block devices (extended volumes) in the extended DOS partition.

If a 05H system ID is found, the start location in that partition entry will be read in order to find the location of the next extended start-up record (extended volume). When located, it will be read in and then the process repeated in order to install additional devices.

Once all the valid devices for a physical drive have been installed, the next physical drive will be examined and the entire process repeated.

A device driver will not assume any order dependency when searching for a particular system ID byte in an extended start-up record. All four possible entries in a extended start-up record partition table will be searched before a driver decides that a particular system ID byte does not exist.

The extended DOS partition can only be created if a Primary DOS or IFS partition already exists on a bootable drive. A Primary DOS partition is a partition with a system id of 01H, 04H, or 06H. A Primary IFS partition is a partition with a system id of 07H. If the driver is not bootable, an extended DOS partition may be created without having a Primary DOS partition. The extended DOS partition starts and ends on a cylinder boundary.

## Creating Block Devices in the Extended DOS Partition

To create the structure for an extended volume in the extended DOS partition, FDISKPM will determine if there is available space in the extended DOS partition and if less than 24 total devices are allocated in the system. The maximum number of block devices allowed is 26, and two are used by diskettes A: and B:. If so, then the program will create an extended start-up record at the space located, with a partition entry filled in with the size and location information for that logical block device. If this is not the first extended start-up record, the program will back up to the last extended start-up record in the chain (as linked by the 05H entries), and create a partition entry in that extended start-up record that has the size and location data for the newly created record. This action will create the pointer required to locate the newly created start-up record.

If this is the first extended start-up record (in the extended DOS partition), only the size, type and location of the logical block device need to be put into a partition entry. The start of the extended DOS partition in the master start-up record will serve as a pointer to this extended volume.

## Deleting Block Devices in the Extended DOS Partition

To delete a block device, the program will set to 0 the 16-byte partition entry that contained the system ID byte that indicated the device type (01H, 04H, 06H, or 07H). Also, if in the same extended start-up record there exists a partition entry with system ID of 05H, indicating that another extended volume exists, this information will be copied to the 05H partition entry of the previous extended start-up record. There is one exception to this rule: if the logical block device deleted is at the beginning of the extended DOS partition, only the partition entry indicating the device type would be set to 0. The 05h pointer information will be left in place.

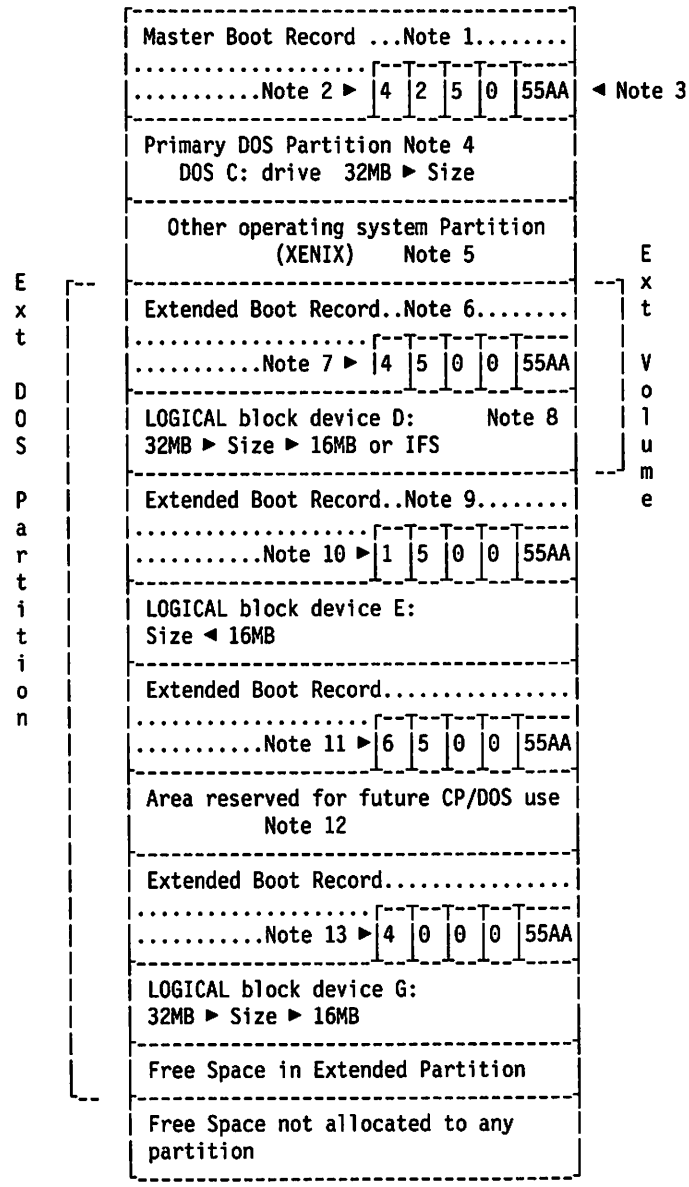



Figure 6-4. Example of layout of large DASD device with Extended DOS partition.

**Note 1** Master start-up (boot) record code, starting at Trk 000, Hd 00, Sec 01 of disk 80H or 81H.

**Note 2** Partition table for master start-up record. See "BPB and Get Device Parameters for Extended Volumes" on page 6-95 for layout. The 4 is the system ID byte in the partition table that indicates a DOS partition greater than 16MB and less than or equal to 32MB. The 2 is a XENIX partition, and the 05H maps the extended DOS partition.

**Note 3** 55AAH is the signature to validate the master start-up record.

**Note 4** Primary DOS area, must reside entirely in first 32MB of disk. C: is block device 80H. D: is block device 81H if it exists. This partition has a maximum size of 32MB.

**Note 5** Other operating system on disk; XENIX in this example.

**Note 6** Extended start-up record for extended volume that corresponds to logical block device D:. (This assumes only 80H block device exists.) If 81H block device exists, this would be block device E:.

**Note 7** Logical block device D: partition table entry. This has a maximum size of 32MB, which is indicated by the system ID of 4. This must set the logical DOS block device as starting at the next track boundary. The 05h system ID byte in the second partition entry maps out the space allocated to the next extended volume. The starting cyl/sec/head in the partition entry with ID of 05H is the location of the next extended start-up record of the next extended volume.

**Note 8** Logical block device D:. Logical DOS devices always begin with a DOS start-up record as does the primary DOS partition.

**Note 9** Extended start-up record for logical block device E:.

**Note 10** Partition table entry for logical block device E:. This logical DOS block device is less than or equal to 16MB, as indicated by the system ID of 01h. The entry with system ID of 05H maps out the space allocated to the next extended volume.

**Note 11** The system ID byte of 06h indicates a logical block device greater than 32MB. This block device is indicated by a block device letter of F. Note also that a pointer exists to the next extended volume.

**Note 12** The greater than 32MB FAT Partition.

**Note 13** Partition table entry for final DOS logical block device. Note the absence of 05h ID byte means that there are no other extended volumes allocated in the extended DOS partition. This would have a block device letter of G: if the previous logical block device was recognized. Otherwise it would be F:.

| Offs | Purpose | Head | Sector | Cylinder |
|---|---|---|---|---|
| 1BE | Partition 1 begin | boot ind | H | S | CYL |
| 1C2 | Partition 1 end | syst ind | H | S | CYL |
| 1C6 | Partition 1 rel sect | Low word | High word |
| 1CA | Partition 1 # sects | Low word | High word |
| 1CE | Partition 2 begin | boot ind | H | S | CYL |
| 1D2 | Partition 2 end | syst ind | H | S | CYL |
| 1D6 | Partition 2 rel sect | Low word | High word |
| 1DA | Partition 2 # sects | Low word | High word |
| 1DE | Partition 3 begin | boot ind | H | S | CYL |
| 1E2 | Partition 3 end | syst ind | H | S | CYL |
| 1E6 | Partition 3 rel sect | Low word | High word |
| 1EA | Partition 3 # sects | Low word | High word |
| 1EE | Partition 4 begin | boot ind | H | S | CYL |
| 1F2 | Partition 4 end | syst ind | H | S | CYL |
| 1F6 | Partition 4 rel sect | Low word | High word |
| 1FA | Partition 4 # sects | Low word | High word |
| 1FE | Signature | |

Figure 6-5. Partition Table for Master Start-up Record

## BPB and Get Device Parameters for Extended Volumes

For purposes of the BIOS Parameter Block (BPB) and Get Device Parameters (Generic IOCtl), an extended volume appears to the system as a virtual physical fixed disk. The extended start-up record will correspond to the master start-up record of a real fixed disk and the logical block device will correspond to the primary DOS partition.

This means that the BPB of the logical DOS block device of the extended volume will describe the environment in the extended volume; this consists of the extended start-up record and the logical block device. The meaning of the fields will be consistent to the meaning of the fields for the primary DOS partition; they relate to the entire physical disk, the primary DOS partition, and the master start-up record. For example, the number of hidden sectors will be the distance from the beginning of the extended start-up record (of the extended volume in question) to the start of the logical DOS block device (the DOS start-up record). The number of sectors field will describe only the logical block device just as it normally only describes the primary DOS partition.

## Category 8 Generic IOCtl Commands

The philosophy described above also applies to the disk generic IOCtl commands. For any logical block device of an associated extended volume; physical cylinder, head, sector I/O is mapped to within the extended volume. Cylinder 0, head 0, sector 1 is mapped to the extended start-up record. An error condition will be generated for any attempt to do C,H,S I/O beyond the size of the extended volume in question.

## Category 9 Generic IOCtl Commands

Category 9 generic IOCtl commands are used to access the entire physical fixed disk without consideration of logical volumes. Physical cylinder, head, sector begin at the start of the physical drive instead of at the beginning of an extended volume. "Get physical device parameters" describes the entire physical device.

## Type 6 Partition

Note the following for a type 6 partition:

- A 12- or 16-bit type FAT could be used to map it since the type of FAT is strictly based on the number of allocation units (clusters) and is the same algorithm used to define the type of FAT (12 vs 16-bit) in OS/2 1.0.

- FAT cluster sizes are defined to be based on powers of 2 as for other DOS partition types. Assuming usage of the OS/2 FORMAT utility, the MINIMUM cluster size for a hard file is 2K. Cluster size and the type of FAT (12 vs 16-bit) is determined by the media partition size. The OS/2 FORMAT algorithm is:

```
If partition size =<16MB
  then do;
   use 12-bit FAT; /* max 4084 entries */
   max cluster size = 4KB;
  end;
  else do;         /* partition size >16MB */
   use 16-bit FAT; /* max 64K entries */
   min cluster size = 2KB;
  end;
```

NOTE however that the actual determination of the partition type is made based on the number of clusters on that partition (OS/2 FORMAT just makes sure that this is true for the < 16MB and > 16MB partitions.

```
.If number of clusters <= 4084
    use 12-bit FAT; /* max 4084 entries */
 else
    use 16-bit FAT; /* max 64K entries */
```

A partition size of 128MB would require a 2K cluster size based on a maximum 64K allocation units (clusters). A partition size of 129MB-256MB would require a 4K cluster size based on 64K allocation units (clusters). A partition size of 257MB-512MB would require an 8K cluster size based on 64K allocation units (clusters).

Configuration table used by OS/2 FORMAT:

| Total # of sectors | Size of partition | Sec/clus | # of root dir entries |
|---|---|---|---|
| 32K | 16M | 8 | 512 |
| 64K | 32M | 4 | 512 |
| 256K | 128M | 4 | 512 |
| 512K | 256M | 8 | 512 |
| 1M | 512M | 16 | 512 |
| 2M | 1G | 32 | 512 |
| 4M | 2G | 64 | 512 |
| 8M | 4G | 128 | 512 |

Please note that for type-6 partitions it is safe to use a non-default configuration, but this may be unsafe for other partition types.

- The partition can reside anywhere on the media, as the primary DOS partition and/or as an extended volume within the extended DOS partition.

- The BPB parameter 'number of sectors per FAT' field width has been extended from a byte to a word in order to define a full 128KB FAT structure. This change affects all DOS partition types.

- In the extended volume context, type 6 > 0 MB.

- In the primary DOS partition context, type 6 > 0 MB. If the partition spans the 32MB boundary or starts at or beyond the 32MB boundary, it can be any size > 0 and **still be a type 6 partition**
- On behalf of a DosDevIoctl 'Get Device Parameters' call;

  If the media is > 32MB, or is defined by a type 6 partition, the recommended BPB must show a 0 in TotalSectors, Ext_TotalSectors must show the number of sectors on the media, and the double word form of 'HiddenSectors' must show the number of sectors after the Master Boot Record to the start of the partition's OS/2 boot record.

**Layout of Block Devices With A Type 6 Partition**

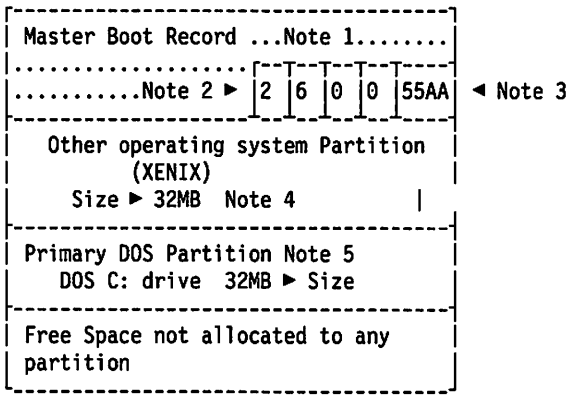



Figure 6-6. Example of layout of large DASD device with a type 6 partition.

**Note 1** Master boot record code, starting at Trk 000, Hd 00, Sec 01 of disk 80H or 81H.

**Note 2** Partition Table for Master boot record. See IBM PC DOS 3.20 Technical Reference for layout. The 2 is the system ID byte in the partition table that indicates a XENIX partition, and the 06H maps a primary DOS type 6 partition.

**Note 3** 55AAH is the signature to validate the master boot record.

**Note 4** Other operating system (XENIX) on disk.

**Note 5** Primary DOS partition. C: is block device 80H, The partition type in this example is a 6 because it ends beyond the first 32MB of the disk. Note that within the scope of this definition that though the size of a primary DOS partition may be less than 32MB, because it ends beyond the first 32MB of the disk it is defined as a type 6.

**Layout of Block Devices With A Type 6 Partition**

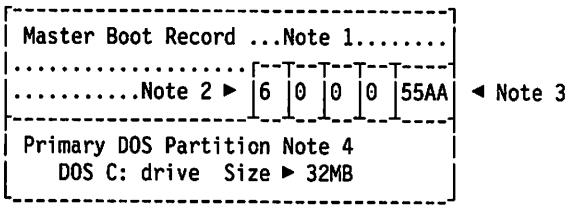



Figure 6-7. Example of layout of large DASD device with type 6 partition.

**Note 1** Master boot record code, starting at Trk 000, Hd 00, Sec 01 of disk 80H or 81H.

**Note 2** Partition Table for Master boot record. See IBM PC DOS 3.20 Technical Reference for layout. The 6 is the system ID byte in the partition table that indicates a DOS partition where 32MB < SIZE.

**Note 3** 55AAH is the signature to validate the master boot record.

**Note 4** Primary DOS area. Owns the entire media and exceeds 32MB in size. C: is block device 80H.

## Type 7 Partition

Note the following for a type 7 partition:

- Partition type 7 are used for Installable File Systems only. The internal FAT file system should not use this partition type, since it will mean that older versions of PC-DOS and OS/2 will not be able to access the partition.

# Video Device Drivers

## Base Video Handlers

Video devices are accessed through the use of Base Video Handlers (BVHs). These BVHs consist of one or more dynamic link libraries (DLLs). In the representative case of the VGA, BVHVGA.DLL manages the device for full-screen sessions, while DISPLAY.DLL (renamed from IBMVGA.DLL) manages the device for the Presentation Manager. Although these device handlers are initialized by different sections of the system at this time, they are architecturally compatible and may easily be combined at a later date.

## Video Device Handler Identification

The list of the active video device handlers and their components resides in CONFIG.SYS as environment variables. To conserve environment space, these variables will be removed from the environment during Shell Initialization.

The value of the VIDEO_DEVICES environment variable is the list of the names of the environment variables which describe each of the video device handlers. Commas are used to separate the names in this list. The following is an example of how you would specify environment variables ARTICHOKE and WATERMELON as those defining the active video device handlers, with ARTICHOKE used as configuration number one and WATERMELON used as configuration number two.

```
SET VIDEO_DEVICES=ARTICHOKE,WATERMELON
```

The value of each environment variable which describes a video device handler is composed of three keywords and the values associated with them. These keywords are separated by blanks and may be specified in any order and in any combination of upper and lower case characters.

The DEVICE() keyword defines the list of names of the dynamic link libraries and device drivers which will be combined to create the video device handler. The names are separated by commas and their order determines the order in which the components will be initialized. These names should represent only those parts of the BVH which need to be called to initialize the Call Vector Table. That is, device drivers should not be included in the list if they are only called by the dynamic link libraries and do not directly modify the Call Vector Table.

The default initialization entry point name for the dynamic link libraries is DEVENABLE. An alternate entry point name may be specified by following any DLL name with +AltName where AltName is the entry point name.

The default initialization IOCTL for device drivers is function 73h category 03h. An alternate function number and category may be specified by following the device name by +Func+Cat where Func is the the function number and Cat is the category. Both numbers must be specified in hexadecimal.

The following is an example of a device handler which is composed of two device drivers, DEV1 and DEV2, and three dynamic link libraries, DYN1 through DYN3. The second device driver uses an alternate initialization IOCTL and the third dynamic link library uses an alternate initialization entry point. The default pointer driver, POINTER$, will be used for both Real Mode and Protect Mode.

```
SET ARTICHOKE=DEVICE(DEV1,DYN1,DEV2+74+05,DYN2,DYN3+OTHERENT)
```

DosOpen will be called for each name in the list to check if it is a device driver, one with an associated DEVICE= statement in CONFIG.SYS. If that call fails, DosLoadModule will be called to check if it is a dynamic link library. If both of these calls fail for any name in the list, the entire device will be ignored.

If the optional PTRDEVR() and PTRDEVP() keywords are specified, they define names of the pointer device drivers to be used in Real Mode and Protect Mode, respectively. If they are not specified, they default to PTRDEVR(POINTER$) and PTRDEVP(POINTER$) The following is an example of a device with only one dynamic link library component and a unique pointer driver for Protect Mode.

```
SET WATERMELON=DEVICE(SEEDLESS) PTRDEVP(PPOINT)
```

**Note:** This design is not limited strictly to physical video devices. By writing your own device handler, you could write video data to the device of your choice, such as a printer or a plotter. Furthermore, by using alternate initialization entry points, multiple devices can be handled by the same physical device handler.

All of the video device handlers shipped with OS/2 are dynamic link libraries. They may be defined by the following environment variables which use the default keywords of PTRDEVP(POINTER$) and PTRDEVR(POINTER$).

```
SET VIO_IBMMPA=DEVICE(BVHMPA)
SET VIO_IBMCGA=DEVICE(BVHCGA)
SET VIO_IBMEGA=DEVICE(BVHEGA)
SET VIO_IBMVGA=DEVICE(BVHVGA)
SET VIO_IBM8514A=DEVICE(BVHVGA,BVH8514A)
SET VIO_IBM8514A=DEVICE(BVH8514A)
```

The following statements would define a system with an 8514 display attached to an 8514/A as the only active video device.

```
SET VIDEO_DEVICES=VIO_IBM8514A
SET VIO_IBM8514A=DEVICE(BVHVGA,BVH8514A)
```

However, the following statement would define a system with an 8514 display attached to an 8514/A and another PS/2 display attached to the VGA connector as independent video devices.

```
SET VIDEO_DEVICES=VIO_IBMVGA,VIO_IBM8514A
SET VIO_IBMVGA=DEVICE(BVHVGA)
SET VIO_IBM8514A=DEVICE(BVH8514A)
```

Two other device handlers are provided with OS/2. The names of these device drivers are fixed. The Base Video Subsystem loads them automatically as they are needed.

BVHINIT.DLL is the Generic Device Handler used by System Installation and System Initialization. It provides the minimum function necessary to support installation of the system and reporting of system errors while booting. It will be loaded only if no other BVHs are successfully loaded.

BVHWNDW.DLL is the Windowable Device Handler which provides the interface to the Presentation Manager by treating PM as a virtual video device. It will be loaded only for Windowed VIO sessions.

## Video Device Chaining

Video device handlers (BVHs) may be chained together when multiple BVHs share the responsibility of supporting a specific video adapter. This is done by allowing previously loaded BVHs to attempt the handling of BVH functions. BVH8514A is the only shipped BVH that provides this support.

During system initialization, BVHVGA is first called to to initialize the Call Vector Table, the table that BVS uses to give control to the BVH routines. BVHVGA initializes as though no other BVH will be handling the device. Next BVH8514A is called to initialize the same Call Vector Table. However, BVH8514A saves a copy of the Call Vector Table before changing it.

When calls are made to this chained BVH, BVH8514A receives the call and passes it to the BVHVGA routine via the saved Call Vector Table. If an error occurs, or if the results of the routine need to be modified, BVH8514A will handle the call. Thus, BVH8514A uses the BVHVGA routines to perform all common functions.

Device chaining can be viewed as a mechanism to allow one BVH to filter function calls to another BVH.

## Primary Display Identification

The primary display is the default display chosen by VIO for full-screen sessions. It is also the display on which VIO and hard error popups are shown. Note that the primary display used by VIO is not necessarily the display on which the Presentation Manager environment runs. The presentation manager would normally run on the highest resolution display. In a dual display configuration, the highest resolution display will not necessarily be a display which can be used in text mode.

The primary display is the pop-up display. A video device driver must determine if it represents the pop-up display. If so, the video device driver must specify that it represents the pop-up display configuration in the Return Config Info call.

## WrtToScrn/Panic Write Support

Before the video subsystem is loaded and when the system is about to abnormally terminate, messages are sent to the screen by the WrtToScrn function, also known as Panic Write. This function switches to Real Mode and executes INT 10h function to set the video mode to a text mode (Bios mode 3). It then uses the Bios INT 10h Write TTY function to display the message.

However, adapters such as the 8514/A have Native modes which cannot be changed by INT 10h. In such cases, the BVH must include a level 0 device driver which hooks INT 10h to provide extended set mode support. This hook must force the adapter out of its native mode, then pass control to the previous INT 10h support.

If these conditions are not satisfied, the adapter should not drive the power up display.

## Running System Installation

The generic device handler, BVHINIT.DLL, is primarily used by System Installation. It is also used for those situations when no video devices have been identified in CONFIG.SYS. It provides only the functions required for System Installation and is otherwise device independent.

Through VioGetConfig, it reports the highest function video adapter and display that it can identify. That is, it can identify only the MPA, CGA, EGA, VGA, and 8514A, along with their respective displays. Although it does not support mode and font setting, it attempts to load the 850 codepage for the current EGA or VGA mode during System Initialization.

If no video devices have been identified in CONFIG.SYS, the generic device handler is loaded by trying to load each of the following devices until one is successfully loaded:

```
GENERIC=DEVICE(BVHVGA,BVH8514A)
GENERIC=DEVICE(BVHVGA)
GENERIC=DEVICE(BVHEGA)
GENERIC=DEVICE(BVHCGA)
GENERIC=DEVICE(BVHINIT)
```

## Using Loadable Device Drivers

Loadable device drivers present a unique problem when used to support video devices, because some video support may be required before the DEVICE= statements have been processed by System Initialization. To handle this problem, two different initialization calls will be made.

The Enable Subfunction determines which type of initialization is to be done. If the subfunction is 1 (Fill Logical Device Block), the DevEnable function is requested to add all of the functions supported by the device handler to the Call Vector Table. If the subfunction is 3 (Fill Initialization Device Block), the InitEnable function is requested to add only those functions which can be supported without the use of a loadable device driver to the Call Vector Table.

Regardless of the success of the InitEnable function, the DevEnable function will be called at Shell Initialization time. Refer to the descriptions of the DevEnable and InitEnable functions for more information.

## Video Device Handler Interfaces

The following functions are video primitives reserved for use by OS/2 components. Unexpected results may occur if these functions are invoked by applications.

All of the video device handler functions described below, except for the DLL Initialization function, use the same calling sequence. Parameters are passed to the routines on the stack. The entry point is found in the BVH Call Vector Table at the index of the function number. The following is the calling sequence used to invoke all the routines:

```
PUSH@   OTHER   Environment       ; Environment Buffer
PUSH@   OTHER   ParmBlock         ; Parameter Block
PUSH    DWORD   Function          ; Function Number

CALL    FAR     BVH Routine Entry Point
```

**where** Environment is the environment buffer. The format of this buffer is defined by the BVH developer. The selector is a huge selector.

ParmBlock is a data structure containing all of the parameters the operation to be performed. The general format of this structure is:

```
Length of parameter block in bytes (=NN)  Word

Flags                                     Word

Bit 0 indicates whether the physical
hardware should be updated.

    Off Initialize only the environment
        buffer
    On  Initialize the environment buffer
        and the hardware state

Bits 1 thru 15 are reserved and must be Off.
```

The length is the total length of the parameter block including the length field itself. The number in parentheses (=NN) represents the value of this field if it is a constant.

The first flag bit always indicates a background vs. foreground state for this function. If the bit is On, the adapter is actually updated, as it is when an application is in the foreground. If the bit is Off, only the buffer that is used to shadow the adapter when an application is in the background is actually updated.

All bits that are not currently defined as reserved must be Off.

Function is the function identifier for this routine. This corresponds to offset into the Call Vector Table and can be used to determine the number of parameters on the stack. This is consistent with the existing DDI used by Presentation Manager.

All routines are expected to return with AX=0 if no error was detected. Otherwise, an error code is returned in AX. Errors common to all commands are:

```
   ERROR_VIO_INVALID_LENGTH if the parameter block length is
incorrect.

   ERROR_VIO_INVALID_PARMS if a reserved flag bit is non-zero or if
   the function number does not match the routine.
```

# DLL Initialization

## DevEnable

This function fills the entries in the Call Vector Table for all of the functions supported by this BVH. It is called as a subfunction of the Winthorn enable function entry point.

**Purpose** Initialize the Call Vector Table for Dynamic Link Libraries

```
PUSH  DWORD Parameter2    ; Variable parameter 2
PUSH  DWORD Parameter1    ; Variable parameter 1
PUSH  DWORD Subfunction   ; Enable subfunction

CALL  FAR   DevEnable
```

**where** Parameter2 for Subfunction=1 is a far pointer to this structure:

```
DWORD Flags Pointer
DWORD Call Vector Table
```

Flags Pointer is a pointer to where the flags controlling calls to the Fill Physical Device Block function go.

Call Vector Table is the pointer to the default dispatch table containing the addresses of the default handler functions and the functions supported by this component.

Each entry in the Call Vector Table is the far address of a Video Device Handler function which must be callable from both ring 2 and ring 3. The far address of the Nth BVH function is the Nth DWord in the table, beginning with function 0.

The functions in the table are defined as follows:

```
000h-0FFh (000-255) Reserved for Presentation Manager
100h-11Fh (256-287) Reserved for BVS
120h-12Fh (288-303) Reserved for IBM JAPAN
130h-13Fh (304-319) Reserved for MSKK

    100h   (256)   Text Buffer Update
    101h   (257)   Initialize Environment
    102h   (258)   Save Environment
    103h   (259)   Restore Environment
    104h   (260)   Return Config Info
    105h   (261)   Get DBCS Display Info
    106h   (262)   Get Color Lookup Table
    107h   (263)   Set Color Lookup Table
    108h   (264)   Get Cursor Info
    109h   (265)   Set Cursor Info
    10Ah   (266)   Get Font
    10Bh   (267)   Set Font
    10Ch   (268)   Get Mode
    10Dh   (269)   Set Mode
    10Eh   (270)   Get Palette Registers
    10Fh   (271)   Set Palette Registers
    110h   (272)   Get Phys Buf
    111h   (273)   Free Phys Buf
    112h   (274)   Get Variable Info
    113h   (275)   Set Variable Info
    114h   (276)   Terminate Environment
    115h   (277)   Print Screen
    116h   (278)   Write TTY
    117h   (279)   Get LVB Info
```

Parameter1 for Subfunction = 1 is a far pointer to this structure:

```
DWORD   Engine Version
DWORD   Count of Table Functions
```

Engine Version is the version of the Presentation Manager Graphics Engine.

Count of Table Functions is the number of entries in the passed dispatch table. The driver can only write this many entries into the table.

Subfunction is the Winthorn enable subfunction number. Its value must be 1 to invoke the Winthorn Fill Logical Device Block subfunction.

All pieces of the BVH must support this function. Pieces of the BVH that do not support the Presentation Manager do not need to support the other functions. Any function not supported should return PMERR_DEV_FUNC_NOT_INSTALLED.

**Remarks** This function is supported by the video dynlinks. This function will only be called once for each adapter to be supported by the driver.

A video device handler should determine if the display adapter and which it supports is present. If not, this function should return an error. Every part of a BVH must successfully initialize the Call Vector Table for that device to be usable by OS/2.

## InitEnable

This function fills the entries in the Call Vector Table for all of the functions supported by this BVH using only SCREEN$ device driver. It is called with parameters similar to the DevEnable entry point specified above, except for the Subfuntion parameter.

**Purpose** Initialize the Call Vector Table for Dynamic Link Libraries

```
PUSH  DWORD Parameter2    ; Variable parameter 2
PUSH  DWORD Parameter1    ; Variable parameter 1
PUSH  DWORD Subfunction   ; Enable subfunction

CALL  FAR   DevEnable
```

**where** Parameter2 for Subfunction = 3 is the same as for the DevEnable function above.

Parameter1 for Subfunction = 3 is the same as for the DevEnable function above. Subfunction is the Winthorn enable subfunction number. Its value must be 3 to invoke the Winthorn Fill Initialization Device Block subfunction. The default entry point of DEVENABLE may be overridden by specifying an alternate name in the DEVICE() parameter describing this BVH in CONFIG.SYS.

**Remarks** This function is called only if video functions are required before Shell Initialization.

The BVH must be able to support the following calls:

```
100h   (256)   Text Buffer Update
101h   (257)   Initialize Environment
102h   (258)   Save Environment    (settable environment only)
103h   (259)   Restore Environment(settable environment only)
104h   (260)   Return Config Info
105h   (261)   Get DBCS Display Info
108h   (264)   Get Cursor Info
109h   (265)   Set Cursor Info
10Ch   (268)   Get Mode            (80x25 text or equivalent only)
10Dh   (269)   Set Mode            (80x25 text or equivalent only)
112h   (274)   Get Variable Info   (codepage only)
113h   (275)   Set Variable Info   (codepage only)
```

**Remarks** See DevEnable section above.

## Text Buffer Update

The Text Buffer Update routine performs text updates to the logical and physical video buffers. All references to the buffer are via the text row and column of each cell affected by the call.

**Function Number** 100h (256)

**Parameter Block format**

```
SIZE   DESCRIPTION

WORD   ParmLength
WORD   Flags
DWORD  AppDataAddr
DWORD  SecondAddr
WORD   Index
WORD   StartRow
WORD   StartCol
WORD   SecondRow
WORD   SecondCol
WORD   RepeatFactor
WORD   LogicalBufSel
WORD   TouchXLeft
WORD   TouchYTop
WORD   TouchXRight
WORD   TouchYBottom
WORD   LVBRowOff
WORD   LVBColOff
WORD   LVBWidth
WORD   LVBHeight
BYTE   LVBFormatID
BYTE   LVBAttrCount
```

ParmLength is the length of the data structure in bytes (> = 26) including the length field itself. The maximum length is 44 bytes. Values not passed are assumed to be the default values for the environment

Flags is defined as follows:

```
Bit 0 indicates whether the physical video buffer needs
to be updated.

Off = The physical video buffer must not be updated.
On  = The physical video buffer must be updated.

Bit 1 indicates whether the logical video buffer needs
to be updated.

Off = The logical video buffer may optionally be updated.
On  = The logical video buffer must be updated.

Bit 2 indicates that attribute information in the user
buffer is in CGA format and may need to be translated
into the format used by the device.  This bit is set
for VioWrtTTY calls any time that ANSI is active,
since ANSI only recognizes CGA format attributes.

Off = Use attributes in existing format
On  = Translation to or from CGA format required

Bits 3 thru 15 are reserved and must be Off.
```

Selector/Offset of the Application Data Area is the pointer to the application's data area which provides either the source or the destination for the buffer operation.

Selector/Offset of Secondary Data Area is the pointer to the additional parameter required by this type of update operation. It is used to define at most one cell of information which is used repetitively as filler. This is used only with Index = 3, 4, 5, 6, or 9.

Index is defined as follows:

```
0 = Read Cell Types from (Row,Col) as word of
    flags:

    Bit 0 indicates whether the cell is part of
    a single or double cell character.

    Off = The cell represents a single cell
          character.
    On  = The cell represents part of a double
          cell character.  Bit 1 should be examined
          for more information.

    Bit 1 indicates whether the cell is a trailing
          cell of a double cell character.

    Off = The cell represents the leading or only
          cell of a character.
    On  = The cell represents the trailing cell of
          a double cell character.

    Bits 2 thru 15 are reserved and must be Off.

 1 = Read Characters from (Row,Col)
 2 = Read Cells from (Row,Col)
 3 = Scroll (Row,Col) through (Row2,Col2) Up
 4 = Scroll (Row,Col) through (Row2,Col2) Down
 5 = Scroll (Row,Col) through (Row2,Col2) Left
 6 = Scroll (Row,Col) through (Row2,Col2) Right
 7 = Write Cells to (Row,Col)
 8 = Write Characters to (Row,Col)
 9 = Write Characters with Constant Attr to (Row,Col)
10 = Write Repeated Character to (Row,Col)
11 = Write Repeated Attribute to (Row,Col)
12 = Write Repeated Cell to (Row,Col)
13 = Copy LVB Rect to PVB
```

Starting Row/Column defines the text location in the Video Buffer at which the update is to be started. For function 13, this is the upper left corner of the rectangle to be written.

Secondary Row/Column defines the text location in the Video Buffer to which cells will be moved. This is used only when moving cells from one location to another (Index=3-6). For function 13 this is the lower right corner of the rectangle to be written.

Repeat Factor is the repeat factor used when updating the buffer. It represents the number of character cells to be updated or the number of rows or columns to be scrolled. This is used for both input and output.

Logical Buffer Selector is the selector used for the beginning of the logical video buffer. This selector is a huge selector with a maximum size of 1MB, even if it is smaller than 64KB.

TouchXLeft is the x coordinate of the upper left corner of the tightest rectangle that circumscribes the cells touched by the current function. If no cells were touched (as in a read), -1 should be returned. Collectively, this field and the next 3 form an area of influence for the call. This field is returned by the BVH.

TouchYTop is the y coordinate of the upper left corner of the tightest rectangle that circumscribes the cells touched by the current function. If no cells were touched (as in a read), -1 should be returned. This field is returned by the BVH.

TouchXRight is the x coordinate of the lower right corner of the tightest rectangle that circumscribes the cells touched by the current function. If no cells were touched (as in a read), -1 should be returned. This field is returned by the BVH.

TouchYBottom is the y coordinate of the lower right corner of the tightest rectangle that circumscribes the cells touched by the current function. If no cells were touched (as in a read), -1 should be returned. This field is returned by the BVH.

LVBRowOff is the row offset of the upper left corner of the LVB in PVB coordinates. All reads, writes and scrolls are done in PVB coordinates.

LVBColOff is the column offset of the upper left corner of the LVB in PVB coordinates. All reads, writes and scrolls are done in PVB coordinates.

LVBWidth is the width of the LVB in cells. This must be > 0.

LVBHeight is the height of the LVB in cells. This must be > 0.

LVBFormatID is the format ID of the LVB. If this value and attribute count are both 0, the Format ID and attribute count for the current mode are used.

LVBAttrCount is the attribute count for the LVB.

**Remarks** The Touchxxx fields for a rectangle that circumscribes the area of the LVB of PVB that was potentially changed by the given operation. For example: a write that included the cells (10,12) to (79,12), (0,13) to (79,13) and (0,14) to (8,14) should return TouchXLeft=0, TouchYTop=12, TouchXRight=79 and TouchYBottom=14.

Any new functions added to the BVH interface that may affect the data in the PVB or LVB must include a return area for the rectangle of the video buffer that was affected by the given call.

The LVBxxx fields indicate that an an LVB of a format possibly differing from the normal LVB for the screen group. If these fields are included, the information about the LVB should be taken from these fields. Note that these fields allow an LVB to begin at a location other than (0,0) and allow LVBs of differing row and column dimensions.

The Text Buffer Update routine returns with AX=0 if no error was detected. Otherwise, the following error codes are returned in AX:

```
    ERROR_VIO_COL if an invalid column number was specified
    ERROR_VIO_INVALID_LENGTH if the ParmLength was
incorrect     ERROR_VIO_INVALID_PARMS if the Index was
incorrect
    ERROR_VIO_MODE if updates are not supported in the current video mode
    ERROR_VIO_ROW if an invalid row number was specified
```

## Initialize Environment

The Initialize Environment routine causes the Environment Buffer, and optionally the video adapter, to be initialized.

**Function Number** 101h (257)

**Parameter Block format**

```
Length of parameter block in bytes (=6)   Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be updated.

        Off Initialize only the environment
            buffer
        On  Initialize the environment buffer
            and the hardware state

    Bit 1 indicates whether the 3xBox is being
    initialized.

        Off The 3xBox is not being initialized.
        On  The 3xBox is being initialized.

    Bits 2 thru 15 are reserved and must be Off.

Logical Video Buffer selector             Word

This selector is a huge selector.
```

**Remarks** It must be possible to call Restore Environment before Save Environment.

This routine always returns with AX=0.

## Save Environment

The Save Environment function is used to save all aspects of the video adapter, including the hardware state and the video buffers.

**Function Number** 102h (258)

**Parameter Block format**

```
Length of parameter block in bytes (=6)   Word
(passed on input)

Flags                                     Word

    Bit 0 is reserved and must be Off.

    Bit 1 indicates whether hardware state
    (mode, CLUT, everything except the buffer)
    should be saved.

    Off = The hardware state should not be saved.
    On  = The hardware state should be saved.

    Bit 2 indicates whether the physical display
    should be fully saved for session switching.

    Off = The physical display should not be
          fully saved.
    On  = The physical display should be
          fully saved.

    Bit 3 indicates whether the physical display
    should be partially saved for popups.

    Off = The physical display should not be
          partially saved.
    On  = The physical display should be
          partially saved.

    Bits 4 thru 15 are reserved and must be Off.

Logical Video Buffer selector             Word

This selector is a huge selector.
```

**Remarks** Bits 2 and 3 are mutually exclusive. If both are specified, bit 3 will be ignored. Bit 4 is used in combination with bits 2 and 3.

The code and data segments referenced or accessed to perform the functions selected by bits 1 and 3 must be locked during device driver initialization.

The format of the data saved in the segments passed as input is determined by the device handler.

Partial saves are invoked on VIO and hard error popups. Popups are displayed on the primary display configuration. The device driver must save whatever portion of the physical display buffer will be overlaid by the popup. To display a popup, OS/2 switches to the highest resolution 80x25 text mode supported by the primary display configuration (mode 3 or 7, whichever is listed first in the list of modes supported by the display configuration). Alternatively, if a device driver's physical display buffer will not be overlaid by a popup, the device driver should return zero for partial save size.

When a hard error popup occurs before a VIO popup has cleared, the Save Environment function will be called twice before the Restore Environment is called. Therefore, the device handler must be prepared to handle both a partial save of a graphics mode and a full save of the text mode of the user popup.

OS/2 allocates the buffer in which the physical display buffer should be saved using DosAllocHuge. The selector to the data packet addresses the first of N segments in which the physical display buffer should be saved. (The offset to the data packet should be ignored.) The selector to the second segment can be calculated by adding the DosAllocHuge increment to the first selector value. The third selector can similarly be calculated by adding the DosAllocHuge increment to the second selector value, etc. Enough selectors are allocated to meet the full/partial buffer requirement specified by the device driver. The selectors each address 64KB except the last selector which addresses the remainder.

The Save Environment routine returns with AX=0 if it can successfully save the environment to the Environment Block and the Logical Video Buffer. Otherwise, it returns with AX = ERROR_VIO_MODE.

## Restore Environment

The Restore Environment function is used to restore all aspects of the video adapter, including the hardware state and the video buffers.

**Function Number** 103h (259)

**Parameter Block format**

```
Length of parameter block in bytes (=6)   Word
(passed on input)

Flags                                     Word

    Bit 0 is reserved and must be Off.

    Bit 1 indicates whether hardware state
    (mode, CLUT, everything except the buffer)
    should be restored.

    Off = The hardware state should not be restored.
    On  = The hardware state should be restored.

    Bit 2 indicates whether the physical display
    should be fully restored for session switching.

    Off = The physical display should not be
          fully restored.
    On  = The physical display should be
          fully restored.

    Bit 3 indicates whether the physical display
    should be partially restored for popups.

    Off = The physical display should not be
          partially restored.
    On  = The physical display should be
          partially restored.

    Bits 4 thru 15 are reserved and must be Off.

Logical Video Buffer selector             Word

This selector is a huge selector.
```

**Remarks** Refer to the Save Environment function for more information.

The Restore Environment routine returns with AX=0 if it can successfully restore the environment from the Environment Block and the Logical Video Buffer. Otherwise, it returns with AX = ERROR_VIO_MODE.

## Return Config Info

The Return Config Info function returns all of the information necessary to identify the current video adapter and display.

**Function Number** 104h (260)

**Parameter Block format**

```
Length of parameter block in bytes (=8)   Word
(passed on input)

Flags -- must be zero                     Word

Far Address of the Config Data            DWord
structure as defined by VioGetConfig.
```

**Remarks** The Environment Buffer is not used by this function. The Environment Buffer address should be passed as a DWORD of zero.

If the length specified in the Config Data is larger than the maximum possible length OR if the length is specified as 2 (the length of length field itself, it will be replaced by the largest valid length. This routine returns with AX=ERROR_VIO_INVALID_LENGTH only if the length specified is less than 2.

## Get DBCS Display Info

The Get DBCS Display Info function returns various forms of DBCS information used by the display.

**Function Number** 105h (261)

**Parameter Block format**

```
Length of parameter block in bytes        Word

    If the length is specified as 2, only
    the maximum length of the parameter
    block will be returned in the length
    field.  If the length is not 2, it
    defines the maximum amount of data to
    be returned.

Flags -- must be zero                     Word

Length of double cell character table     Word

Offset of double cell character table     Word

    The double cell character table consists
    of pairs of words which define the low and
    high limits (inclusive) of ranges of double
    cell characters.
```

**Remarks** This function is used to get the DBCS display information associated with the given environment buffer.

This routine returns with AX=ERROR_VIO_INVALID_LENGTH if the length specified is less than 2 or the buffer was too short to return all of the DBCS display information. Otherwise Get DBCS Display Info routine returns with AX=0.

## Get Color Lookup Table

The Get Color Lookup Table function reads the definitions of the colors from the Color Lookup Table

**Function Number** 106h (262)

**Parameter Block format**

```
Length of parameter block in bytes (=12)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be read.

        Off Return data from the environment
            buffer only.
        On  Read the hardware to update
            the environment buffer before
            returning the requested data.

    Bits 1 thru 15 are reserved and must be Off.

Far address of the color lookup table     DWord

    The format of the table is device
    dependent.  Three-byte table entries
    will be returned for the VGA.  Each
    table entry contains a red, green and
    blue color index, respectively.

Index of first table entry to get         Word

Number of table entries to get            Word
```

The Get Color Lookup Table routine returns with AX=0 if it can successfully get all of the requested registers from the color lookup table. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS

if an invalid color register was requested or AX = ERROR_VIO_INVALID_LENGTH if too many registers were requested.

## Set Color Lookup Table

The Set Color Lookup Table function loads the definitions of the colors from the Color Lookup Table

**Function Number** 107h (263)

**Parameter Block format**

```
Length of parameter block in bytes (=12)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be updated.

        Off Update only the environment
            buffer
        On  Update the environment buffer
            and the hardware state

    Bits 1 thru 15 are reserved and must be Off.

Far address of color lookup table         DWord

    The format of the table is device
    dependent.  The VGA format has three
    bytes containing the red, green and
    blue indices, respectively, for each
    color being set.

Index of first table entry to set         Word

Number of table entries to set            Word
```

The Set Color Lookup Table routine returns with AX=0 if it can successfully set all of the registers in the color lookup table. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS if an invalid color register was requested or AX = ERROR_VIO_INVALID_LENGTH if too many registers were requested.

## Get Cursor Info

The Get Cursor Info routine gets all of the information related to the cursor.

**Function Number** 108h (264)

**Parameter Block format**

```
Length of parameter block in bytes (=16)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be read.

        Off Return data from the environment
            buffer only.
        On  Read the hardware to update
            the environment buffer before
            returning the requested data.

    The remaining flags select the information
    to be returned:

    Bit 1 selects cursor position

    Bit 2 selects cursor type

    Bits 3 thru 15 are reserved and must be Off.

Row (where 0 is the top row)              Word

Column (where 0 is the left column)       Word

Top cursor scan line.  If n scan          Word
lines, 0 is top scan line and n-1 is
bottom scan line.)

Bottom cursor scan line.  If n scan       Word
lines, 0 is top scan line and n-1 is
bottom scan line.)

Cursor width (in columns if text          Word
mode, in pels if graphics mode)

Cursor attribute (-1 = hidden if          Word
text mode, any other value = color
attribute if graphics mode)
```

**Remarks** The Get Cursor Info routine returns with AX=0 if it can successfully get all of the cursor information requested. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

## Set Cursor Info

The Set Cursor Info routine sets all of the information related to the cursor.

**Function Number** 109h (265)

**Parameter Block format**

```
Length of parameter block in bytes (=16)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be updated.

        Off Update only the environment
            buffer
        On  Update the environment buffer
            and the hardware state

    The remaining flags select the options
    to be set:

    Bit 1 selects cursor position

    Bit 2 selects cursor type

    Bits 3 thru 15 are reserved and must be Off.

Row (where 0 is the top row)              Word

Column (where 0 is the left column)       Word

Top cursor scan line.  If n scan          Word
lines, 0 is top scan line and n-1 is
bottom scan line.)

Bottom cursor scan line.  If n scan       Word
lines, 0 is top scan line and n-1 is
bottom scan line.)

Cursor width (in columns if text          Word
mode, in pels if graphics mode)

Cursor attribute (-1 = hidden if          Word
text mode, any other value = color
attribute if graphics mode)
```

The Set Cursor Info routine returns with AX=0 if it can successfully set all of the cursor information requested. Otherwise, it returns with AX = :

```
ERROR_VIO_MODE if it cannot support the function in the current mode.
ERROR_VIO_ROW if the row number is out of range.
ERROR_VIO_COL if the column number is out of range.
```

## Get Font

The Get Font function returns the current active font or a selected font for the current codepage. The format of the font definition is determined by the type of adapter.

**Function Number** 10Ah (266)

**Parameter Block format**

```
Length of parameter block in bytes (=14)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be read.

        Off Return data from the environment
            buffer only.
        On  Read the hardware to update
            the environment buffer before
            returning the requested data.

    Bit 1 indicates whether a specific
    font should be returned instead of the
    current font.

        Off Return the current font.
        On  Return the selected font for
            the current codepage.  Setting
            this flag indicates that the
            pel columns and rows will be used
            as input to select the font.

    Bits 2 thru 15 are reserved and must be Off.

Far address of the font buffer            DWord

    Data area in which the font definition
    will be returned.

Length of data area in which font         Word
table should be returned.

Pel columns                               Word

Pel rows                                  Word
```

**Remarks** If the Length is specified as 0, no font will be returned. Instead, the length field will return the size needed to hold the font.

The Get Font routine returns with AX=0 if it can successfully read the font. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

## Set Font

The Set Font function sends a user font definition to the device handler. If the font is appropriate for the current mode, it is loaded into the adapter. If not, it is saved for possible use on subsequent SetMode calls. The format of the font definition is determined by the type of adapter.

**Function Number** 10Bh (267)

**Parameter Block format**

```
Length of parameter block in bytes (=14)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be updated.

        Off Update only the environment
            buffer
        On  Update the environment buffer
            and the hardware state

    Bits 1 thru 15 are reserved and must be Off.

Far address of the font buffer           DWord

    The font buffer contains the font
    to be set (in compact form).

Length of data area containing the        Word
font table to be set

Pel columns                               Word

Pel rows                                  Word
```

The Set Font routine returns with AX=0 if it can successfully load the font. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

## Get Mode

The Get Mode function returns all of the information pertaining to the current video mode.

**Function Number** 10Ch (268)

**Parameter Block format**

```
Length of parameter block in bytes (=8)   Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be read.

        Off Return data from the environment
            buffer only.
        On  Read the hardware to update
            the environment buffer before
            returning the requested data.

    Bits 1 thru 15 are reserved and must be Off.

Far Address of the Mode Data structure    DWord
as defined by VioGetMode.
```

If the length specified in the Config Data is larger than the maximum possible length OR if the length is specified as 2 (the length of length field itself, it will be replaced by the largest valid length.

This routine returns with AX=ERROR_VIO_INVALID_LENGTH only if the length specified is less than 2.

## Set Mode

The Set Mode routine sets the video mode of the video adapter. For text modes, it considers not only what display characteristics it can support, but also what ROM, codepage, and user defined fonts it has available.

**Function Number** 10Dh (269)

**Parameter Block format**

```
Length of the data structure in bytes     Word
including the length itself (=8).

Flags                                     Word

    Bit 0  indicates whether the physical
    hardware should be updated.

        Off Update only the environment
            buffer
        On  Update the environment buffer
            and the hardware state

    Bit 1  indicates whether the mode
    should be changed or only validated

        Off Perform normal mode setting
        On  Perform only mode validation

    Bits 1 thru 15 are reserved and must be Off.

Far Address of the Mode Data structure   DWord
as defined by VioSetMode.
```

**Remarks** This function must validate the mode data without using the environment buffer because it may not have been initialized or may not be valid for this device. This function will implicitly initialize the Environment Buffer if it has not already been done.

The Set Mode routine returns with AX=0 if it can set the requested mode. Otherwise, it returns with AX = ERROR_VIO_MODE.

## Get Palette Registers

The Get Palette Registers function queries the relationship between the text attributes and the color registers.

**Function Number** 10Eh (270)

**Parameter Block format**

```
Length of parameter block in bytes (=12)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be read.

        Off Return data from the environment
            buffer only.
        On  Read the hardware to update
            the environment buffer before
            returning the requested data.

    Bits 1 thru 15 are reserved and must be Off.

Far address of the palette buffer         DWord

    Data area in which a one-word entry
    for each register containing its
    color value will be returned.

Index of first palette register to        Word
get

Number of registers to get                Word
```

The Get Palette Registers routine returns with AX=0 if it can successfully get all of the requested palette registers. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS if an invalid color register was requested or AX = ERROR_VIO_INVALID_LENGTH if too many registers were requested.

## Set Palette Registers

The Set Palette Registers function defines the relationship between the text attributes and the color registers.

**Function Number** 10Fh (271)

**Parameter Block format**

```
Length of parameter block in bytes (=12)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be updated.

        Off Update only the environment
            buffer
        On  Update the environment buffer
            and the hardware state

    Bits 1 thru 15 are reserved and must be Off.

Far address of palette register buffer    DWord

    Data area with one word, containing
    the color value, for each register
    being set.

Index of first palette register to set    Word

Number of registers to set                Word
```

The Set Palette Registers routine returns with AX=0 if it can successfully set all of the requested palette registers. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS if an invalid color register was requested or AX = ERROR_VIO_INVALID_LENGTH if too many registers were requested.

## Get Phys Buf

The Get Phys Buf function returns an LDT selector which can be used to access the physical video buffer. The current physical video buffer is returned unless a specific address range is requested.

**Function Number** 110h (272)

**Parameter Block format**

```
Length of parameter block in bytes (=12)  Word
(passed on input)

Flags -- must be zero                     Word

Far Address of the Get Phys Buf data      DWord
structure as defined by VioGetPhysBuf.
```

**Remarks** If the physical display buffer address and length passed on input are zero, this call returns an LDT selector which corresponds to the current mode.

A video device driver must provide read/write access to the physical address range where the physical display buffer is located. The device driver must provide read only access to the physical address range where the ROM fonts are located. If the physical address passed on input is not within the physical display buffer or ROM font ranges, an error should be returned.

The Get Phys Buf routine returns with AX=0 if it can successfully allocate the LDT selector. Otherwise, it returns with AX set by DevHelp_PhysToUVirt or AX = ERROR_VIO_INVALID_PARMS if requested buffer resides outside the valid range for the device.

## Free Phys Buf

The Free Phys Buf routine deallocates an LDT selector that was acquired using the Get Phys Buf routine.

**Function Number** 111h (273) dt.Parameter Block format

| | |
|---|---|
| Length of parameter block in bytes (=6) (passed on input) | Word |
| Flags -- must be zero | Word |
| LDT Selector | Word |

This routine always returns with AX=0.

## Get Variable Info

The Get Variable Info routine reads various minor features of the video adapter, including the blink state, border color, underscore line and scrollable rectangle of the screen.

**Function Number** 112h (274)

**Parameter Block format**

```
Length of parameter block in bytes (=26)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be read.

        Off Return data from the environment
            buffer only.
        On  Read the hardware to update
            the environment buffer before
            returning the requested data.

    The remaining flags select the information
    to be returned:

    Bit 1 selects blink versus background
          color
    Bit 2 selects overscan (border) color
    Bit 3 selects scan line for underscore
    Bit 4 selects video enable
    Bit 5 selects the display mask
    Bit 6 selects code page
    Bit 7 forces a code page set (used with 6)
    Bit 8 gets the scrollable rectangle

    Bits 9 thru 15 are reserved and must be Off.

Blink versus background intensity         Word
  0 = blink
  1 = background intensity

Overscan (border) color                   Word

Scan line for underscore (0-31)           Word
  32 = no underscore

Video enable                              Word
  0 = off
  1 = on

  Off means off until the device
  driver is told to turn it back on.
  If the video signal is turned off
  and then the mode is set, the
  signal must remain off.

Display mask                              DWord

  bit 0  = plane 0

    |      |
    |      |
    |      |

  bit 31 = plane 31

  bit state = 0, plane disabled
                 for display
  bit state = 1, plane enabled
                 for display

  (Planes disabled for display
  result in 0 to palette.)

Code page                                 Word

Scrollable Rectangle - Left               Word
```

```
Scrollable Rectangle - Top                Word

Scrollable Rectangle - Right              Word

Scrollable Rectangle - Bottom             Word
  The scrollable rectangle fields
  indicate the area of the screen
  that may scroll during scroll
  and write TTY operations.

Screen Rows                               Word
  The number of text rows in the
  current mode.

Screen Columns                            Word
  The number of text columns in
  the current mode.
```

The Get Variable Info routine returns with AX=0 if it can successfully get the selected variable information. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

## Set Variable Info

The Set Variable Info routine sets various minor features of the video adapter, including the blink state, border color, and underscore line and scrollable rectangle of the screen.

**Function Number** 113h (276)

**Parameter Block format**

```
Length of parameter block in bytes (=26)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be updated.

       Off Update only the environment
           buffer
       On  Update the environment buffer
           and the hardware state
```

```
    The remaining flags select the options
    to be set:

    Bit 1 selects blink versus background
          color
    Bit 2 selects overscan (border) color
    Bit 3 selects scan line for underscore
    Bit 4 selects video enable
    Bit 5 selects the display mask
    Bit 6 selects code page
    Bit 7 forces a code page set (used with 6)
    Bit 8 sets the scrollable rectangle
    Bits 9 thru 15 are reserved and must be Off.

Blink versus background intensity        Word
  0 = blink
  1 = background intensity

Overscan (border) color                  Word

Scan line for underscore (0-31)          Word
  32 = no underscore

Video enable                             Word
  0 = off
  1 = on

  Off means off until the device
  driver is told to turn it back on.
  If the video signal is turned off
  and then the mode is set, the
  signal must remain off.

Display mask                             DWord

  bit 0  = plane 0

    |      |
    |      |
    |      |

  bit 31 = plane 31

  bit state = 0, plane disabled
                 for display
  bit state = 1, plane enabled
                 for display

  (Planes disabled for display result
  in 0 to palette.)

Code page                                Word

Scrollable Rectangle - Left              Word

Scrollable Rectangle - Top               Word

Scrollable Rectangle - Right             Word

Scrollable Rectangle - Bottom            Word
  The scrollable rectangle fields
  indicate the area of the screen
  that may scroll during scroll
  and write TTY operations.

Screen Rows (Reserved and must be 0)     Word
  The number of text rows in the
  current mode.

Screen Columns (Reserved and must be 0)  Word
  The number of text columns in
  the current mode.
```

There are two types of code page sets. The first code page set allows a codepage to be set while the mode of the display (as per Set Mode) remains the same. The other type of code page set causes a change in the display mode. This will occur when switching between DBCS and non-DBCS code pages.

If the 6th bit of the flags word is set and the 7th is clear, the codepage should be set if the adapter can use the code page without changing from a DBCS to an SBCS mode or vice versa.

If both the 6th and 7th bits of the flags word are set and the code page can only used if the mode is changed from an SBCS to a DBCS mode, then the mode should be changed.

The 7th bit is ignored if the 6th bit is not set. Also, this only applies to text modes. Graphics modes are not set to text modes by forcing a code page.

The BVH need not support any scrollable region other than the entire display area. The adapter may support any scrollable rectangle up to the entire screen. All coordinates are in text display cells. This scrollable rectangle data is undefined for graphics modes.

The Set Variable Info routine returns with AX=0 if it can successfully set the selected variable information. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

## Terminate Environment

The Terminate Environment function is used to notify the BVH that the environment is about to be freed so that any required cleanup may be performed by the BVH.

If no Terminate Environment processing is required, this function may be omitted. If so, PMERR_DEV_FUNC_NOT_INSTALLED will be returned in AX, but will be ignored by the video subsystem.

**Function Number** 114h (276)

**Parameter Block format**

```
Length of parameter block in bytes (=6)   Word
(passed on input)

Flags                                     Word

    All flags are reserved and must be Off.

Logical Video Buffer selector             Word

    This selector is a huge selector.
```

**Remarks** None.

## Print Screen

The Print Screen routine causes the contents of the current screen to be written to the printer handle provided. BVS provides a default routine which provides the same level of support as version 1.1 if the vector is not replaced.

**Function Number** 115h (277)

**Parameter Block format**

```
Length of parameter block in bytes (=8)   Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    video buffer should be printed.

        Off Print only the contents of the
            logical video buffer.
        On  Print the contents of the physical
            video buffer, if appropriate.

    Bits 1 thru 15 are reserved and must be Off.

Logical Video Buffer selector             Word

    This selector is a huge selector.

Print Device Handle                       Word

    This is the file handle of the print
    device to be used.
```

# Write TTY

The Write TTY routine performs the functions of the VioWrtTTY call. BVS provides a default routine which provides the same level of support as version 1.1 if the vector is not replaced.

**Function Number** 116h (278)

**Parameter Block format**

```
Length of parameter block in bytes (=14)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical video
    buffer needs to be updated.

    Off = The physical video buffer
          must not be updated.
    On  = The physical video buffer
          must be updated.

    Bit 1 indicates whether the logical video
    buffer needs to be updated.

    Off = The logical video buffer
          may optionally be updated.
    On  = The logical video buffer
          must be updated.

    Bit 2 indicates whether ANSI is active.

    Off = ANSI is not active and escape
          sequences should be considered
          as text data.
    On  = ANSI is active and escape sequences
          must be handled locally or passed
          to the default routine in BVS via
          device chaining.
```

Bit 3 indicates whether Ctrl-PrtSc is active.

Off = Ctrl_PrtSc is not active.
On = Ctrl_PrtSc is active and characters need to be echoed to the printer locally or by the default routine in BVS via device chaining.

Bits 4 thru 15 are reserved and must be Off.

Logical Video Buffer selector — Word

This selector is a huge selector.

Far Address of Character String — DWord

This is the address of the character string to be written.

Length of Character String — Word

This is the length of the character string to be written.

Print Device Handle — Word

This is the file handle of the print device to be used for Ctrl-PrtSc.

## Get LVB Info

The Get LVB Info routine gets information associated with the LVB, such as the allocation size and default attribute for a specified LVB.

**Function Number** 117h (279)

**Parameter Block format**

```
Length of parameter block in bytes (=20)  Word
(passed on input)

Flags                                     Word

    Bit 0 indicates whether the physical
    hardware should be read.

        Off Read from the environment
            buffer
        On  Read from the current state
            of the hardware state

    Bits 1 thru 15 are reserved and must be Off.


Format ID for the LVB, if this and        Byte
the attribute count are both 0, the
current mode values should be used.

Attribute Count for the LVB.              Byte

LVB Width in cells.                       Word

LVB Height in cells.                      Word

Allocation size of the LVB is returned    DWord
here.

Size of the default attribute return      Word
buffer

Pointer to the default attribute          DWord
attribute return buffer (passed).
The default attribute is returned
if the buffer is large enough.
If this value is 0, the attribute
is not returned.
```

**Remarks** The Get LVB Info routine returns with AX=0 if it can successfully calculate the LVB size and return the attribute information. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

# Level 0 Device Driver Interfaces

The Strategy portion of the level 0 device drivers that may be a component of any video device handler is called to handle I/O requests through a request packet interface with the OS/2 kernel. The strategy routine executes at task-time as a result of an application VIO request. The strategy routine is called with ES:BX pointing to the request packet (the pointer is valid in both the DOS mode and the OS/2 mode). Only three command codes (passed in the request packet) are required to be supported by a video device driver. For any other command code that the device driver does not wish to support, the video device driver should return 'Unsupported Command' and 'Done' in the request packet status field.

The following are the device driver commands that are supported by the video device driver (the command codes are in parentheses):

### INIT ( 00H ) - Initialize the Device

On entry, the request block contains the following fields as inputs to the video device driver:

- Pointer to the DevHlp entry point
- Pointer to the INIT arguments

On exit, the video device driver will set the first pointer to the offsets of the code and data segments to release code and data needed only by the initialization routine. The second pointer will be set to zero. If initialization is successful, the request packet status field will be set to indicate 'NoError' and 'Done', otherwise the status will be set to 'GeneralFailure'.

The video device driver may perform the following initialization:

- Obtains the DevHlp address from the request packet
- Verify that the display adapter and display which it supports are present. If not, it may fail initialization.

### OPEN ( 0DH ) - Open the Device

This service routine does nothing but return with 'no error' status.

### GENERIC IOCtl ( 10H ) - Send I/O Requests to the Device

On entry, the request packet will have the IOCtl category code and function code set. The parameter buffer and the data buffer addresses will be set as virtual addresses. The video device driver will perform the following functions when requested:

## DD Initialization

This IOCTL supports the same class of functions as the Winthorn dynlink enable entry point. It is called to fill in the Call Vector Table for the BVH. The call should return an error if the driver detects that the adapter is not present.

**Category 3** Function 73H

**Purpose** Initialize Call Vector Table

**Parameter Packet Format** None

**Data Packet Format**

```
DWORD    Subfunction
DWORD    Parameter1
DWORD    Parameter2
WORD     ReturnCode
```

**where** Subfunction is the Winthorn enable subfunction number. Its value must be 1 to invoke the Winthorn Fill Logical Device Block subfunction.

All pieces of the BVH must support this function. Pieces of the BVH that do not support the Presentation Manager do not need to support the other functions. Any function not supported should return PMERR_DEV_FUNC_NOT_INSTALLED.

Parameter1 for Subfunction = 1 is a far pointer to this structure:

```
DWORD   Engine Version
DWORD   Count of Table Functions
```

Engine Version is the version of the Presentation Manager Graphics Engine.

Count of Table Functions is the number of entries in the passed dispatch table. The driver can only write this many entries into the table.

Parameter2 for Subfunction = 1 is a far pointer to this structure:

```
DWORD   Flags Pointer
DWORD   Call Vector Table
```

Flags Pointer is a pointer to where the flags controlling calls to the Fill Physical Device Block function go.

Call Vector Table is the pointer to the default dispatch table containing the addresses of the default handler functions and the functions supported by this component.

Each entry in the Call Vector Table is the far address of a Video Device Handler function which must be callable from both ring 2 and ring 3. The far address of the Nth BVH function is the Nth DWord in the table, beginning with function 0. Refer to the DLL Initialization function for a description of the function numbers,

**Remarks** This function is supported by the video device drivers.

This function will only be called once for each adapter to be supported by the BVH.

A video device handler should determine if the display adapter and which it supports is present. If not, this function should return an error. Every part of a BVH must successfully initialize the Call Vector Table for that device to be usable by OS/2.

# The Printer Device Driver

The OS/2 printer device driver is a base device driver which supports the parallel device interface for OS/2. There are two distinct versions of the printer device driver:

**PRINT01.SYS** supporting parallel port adapters on Family One bus machines.

**PRINT02.SYS** supporting parallel port adapters on machines containing Micro Channel Architecture.

The family one printer device driver manipulates the hardware ports through IN/OUT instructions. The family two printer device driver does not access the hardware ports. Advanced BIOS, which executes on the family two machines, replaces the hardware port manipulation done on the family one machines.

Only one of these device drivers will be resident for each machine running OS/2. Since the printer device driver is a base device driver, OS/2 determines the system's hardware configuration and automatically loads the appropriate device driver when the system is initialized. A DEVICE = command in CONFIG.SYS is not required to install the printer device driver.

## Overview

The printer device drivers are bimodal device drivers. Their function is to service parallel port requests from DOS Mode and OS/2 applications (including those running under the Presentation Manager). Requests are serialized so that mixed printer output will not occur. Printer requests include:

- Initializing a parallel printer device
- Sending characters to a parallel printer device
- Returning the status of a parallel printer device
- Handling IOCTL function calls, including the setting of the printer device's frame control (characters per line and lines per inch).

In addition, the printer device drivers include character device monitor support. The printer device driver will send data to a character device monitor if one is installed.

The primary method of communicating with the printer device driver is through a request packet. When the printer device driver receives a request packet, it determines which printer device the request is for. If a request is not already in progress, the device driver calls an appropriate routine to handle the request. If a request is in progress, the device driver blocks the caller's thread until the in-progress request has completed. Once the in-progress request has been serviced, the device driver will run the caller's thread and process the waiting request.

To make a request of the printer (for example, to write characters to the printer) in the OS/2 mode, an application pushes parameters (for example, the device handle, the address of characters to be written) onto its stack and calls the file system API (for example, DosWrite). The file system:

- Determines that the request is for the printer device driver (based on the specified device handle)
- Creates a request packet
- Issues a call to the printer device driver strategy routine.

To make a request of the printer in the DOS mode, an application moves the parameters into predefined registers and issues an interrupt 17H or an interrupt 21H. The interrupt 17H will enter directly into the printer device driver at the interrupt 17H handler. The interrupt 21H request is converted into a request packet and issued to the device driver strategy routine.

## Replacing the Printer Device Driver

The printer device drivers, both the family one and family two, contain four device headers in their data segments with the reserved device names PRN, LPT1, LPT2 and LPT3 corresponding to the three logical devices supported by this device driver.

**Note:** The names PRN and LPT1, although logically appearing different, correspond to the same physical device.

The printer device driver is replaceable, that is, it is deinstallable by:

- Creating another printer device driver with the device name in the device header identical as one listed above; and then
- Specifying the new printer device driver in a DEVICE= statement in CONFIG.SYS.

  **Note:** Installation of the files associated with the new device driver and the addition of the corresponding DEVICE= statement in CONFIG.SYS is accomplished through the DDINSTAL utility and the DDP (device driver profile) for the new device driver.

When the DEVICE= statement for the new printer device driver is encountered in CONFIG.SYS during system initialization, the system will generate a deinstall request packet and send it to the base printer device driver. The base printer device driver receives the deinstall request packet and releases its resources. See "14H / DEINSTALL Terminate the Device Driver" on page 4-21 in this book for more information.

If the base printer device driver successfully releases its resources it returns a successful completion on the deinstallation request. When the base device driver has been deinstalled, the system then generates an install request packet and sends it to the new device driver. See "0H / INIT Initialize Device Driver" on page 4-5 in this book for more information.

## Code Page Support

The printer device driver provides code page support only through the OS/2 character device monitor mechanism, for example, the print spooler. In addition, three printer IOCTL commands support code page and font switching so that an application has the ability to change the active code page and font in the middle of its print job.

### Printer Monitors

The printer device driver defines a data stream for each physical printer device. In addition, each printer data stream may be monitored by printer device monitor applications, for example, the printer spooler, registered with one of two monitor chains. Character device monitor support is provided by the printer device driver, therefore, for each physical printer device.

To register a monitor for one of the printer devices, an application issues the DosMonOpen and DosMonReg monitor function calls. (See Chapter 8, "Character Device Monitors" in this book for more information.) On behalf of the application, DosMonReg generates an IOCTL request to the printer device driver to register a monitor on one of the two monitor chains associated with the device. (See Category 10, Function 40H.) When the printer device driver receives this request, it:

- Creates a monitor chain for that device by calling the MonitorCreate device helper routine if none was previously created as the result of another register monitor IOCTL request; and
- Registers the monitor with the monitor chain by calling the Register device helper routine.

By setting the DosMonReg index parameter appropriately, the application specifies on which of the two monitor chains its monitors are registered. The first monitor chain (specified by the application by setting index = 1) is considered to be the data chain and is used by the printer device driver to send data to a monitor. The second monitor chain (specified by the application by setting index = 2) is considered to be the code page chain.

To insure that printer requests are processed in order, the printer device driver places all data and code page requests only into the first monitor chain (referred to as the data chain). That is, the printer device driver issues MonWrite device helper calls only to the data chain. The printer device driver does not place data into the code page chain by calling the MonWrite device helper routine.

To improve performance, the second monitor chain (referred to as the code page chain) is used by the printer device driver primarily to receive the results of a code page request. Processes that issue code page requests are blocked until they receive an indication that their request is valid. Since the number of code page requests is negligible compared to the number of data requests, printer monitors can respond more quickly and efficiently to the device driver through the code page chain.

To summarize, printer monitors:

- Receive all data and code page requests from the first monitor chain (see DosMonRead)
- Return data requests to the first monitor chain (see DosMonWrite)
- Return code page requests to the first or the second monitor chain (see DosMonWrite), depending on the position of the monitor when other monitors are registered with the chain (see below).

  **Note:** The printer device driver expects to receive the monitor responses to code page requests on the code page chain. If monitor responses to code page requests are not received from any monitor on the code page chain, unpredictable results will occur.

The print spooler is an example of an application that registers a printer monitor with both the data chain and the code page chain for a printer device. Because no requests are placed into the second monitor chain by the printer device driver, the monitor input buffer specified when the spooler registers the monitor with the second chain (see DosMonReg, with index = 2) is not used.

*Printer monitors must be designed with extreme care. They must be positioned carefully when other printer monitors, in particular the print spooler, are registered:*

- If an application monitor wishes only to process character data, it registers a printer monitor only on the data chain (index = 1) when:

  1. It is the only monitor in the monitor chain (for example, the spooler is not loaded).
  2. It is registered in a position to process the data after the spooler.

     This monitor will never see code page requests from the spooler, because the spooler automatically sends these requests back to the device driver through the code page chain (index = 2).

- If an application wishes to process both character data and code page requests, it registers a printer monitor on the data chain (index = 1) and a printer monitor on the code page chain (index = 2) when:

  1. It is the only monitor in the monitor chain.

     It must expect to receive both data and code page requests on the data chain (index = 1). It must respond to the code page requests on the code page chain (index = 2) as quickly as possible.

  2. It is registered in a position to process data after the spooler.

     It must expect to receive character data from the spooler on the data chain (index = 1) and the code page requests from the spooler on the code page chain (index = 2).

     Because the spooler will pass the code page results along the code page chain before all the data has been spooled and released to be printed, a printer monitor cannot easily synchronize the code page requests with the data requests under these circumstances.

- If a printer monitor wishes to process character data and code page requests and it is registered in a position to process data before the spooler, it registers on the data chain (index = 1) only.

  All data and code page requests will be sent to the first (non-spooler) monitor on the data chain. Because the spooler receives data and code page requests from the data chain only, this monitor must pass on to the spooler through the data chain (index = 1) all the information it receives in the order that it is received.

## IOCTL support of Code Page and Font Switching

The printer device driver has three printer specific IOCTL commands to support the code page and font switching provided in OS/2. In order to support these IOCTL calls, there are Font Monitor Buffer command/responses in the monitor interface between the printer device driver and the spooler.

All of the actual code page and font switching function for printers is provided by the code page switcher (SPOOLCP.DLL). When the spooler is started, it checks to see if code page support is required. If it is, the spooler will cause the code page switcher to be loaded and initialized. The spooler interfaces with the code page switcher through DOSPFS function calls.

The following illustrates the dialog between the printer device driver and the spooler when the printer device driver receives one of these special IOCTL requests from an application:

1. The printer device driver receives an IOCTL request from an application to activate a code page.
2. If the spooler is loaded, the printer device driver sends a Font Monitor Buffer command to the spooler in the form of a special monitor record placed into the data monitor chain (see the MonWrite device helper routine).
3. The spooler receives this special monitor record from its monitor input buffer registered with the data monitor chain by calling DosMonRead.
4. When the spooler receives this special monitor record, it calls DosPFSActivate, to get information from the DCP (Define Code Page) system files. This information is placed into a temporary spooler file.
5. The spooler sends a return code back to the printer device driver by calling DosMonWrite to place a special monitor record (that is, a Font Monitor Buffer response) into its monitor output buffer registered with the code page chain.

The three printer specific IOCTL commands to support code page and font switching are:

*Activate Font*

When an application issues a DosOpen for a printer (that is, PRN, LPT1, LPT2, or LPT3), the file system issues an Activate Font IOCTL to the printer device driver to set the active code page and font according to the active code page of the process making the open request.

At any time an application may also issue the Activate Font IOCTL to the printer device driver by using the handle returned from the DosOpen call to open the device. This allows the application to change the active code page and font in the middle of its print job.

When the printer device driver receives the Activate Font IOCTL, if the spooler or another monitor is registered, the printer device driver sends a Font Monitor Buffer command (in this case, an Activate Font command) to the monitor on the data chain. The monitor receives this special monitor record from its monitor input buffer registered on the data chain by calling DosMonRead. The monitor sends a Font Monitor Buffer response to this command to the printer device driver by calling DosMonWrite, to place a special monitor record into its monitor output buffer registered on the code page chain.

If the spooler, or another monitor, is not registered, the printer device driver returns the appropriate return code to the caller of the Activate Font IOCTL.

*Query Active Font*

When the printer device driver receives the Query Active Font IOCTL request, if the spooler or another monitor is registered, the printer device driver sends a Font Monitor Buffer command (in this case, a Query Active Font command) to the monitor on the data chain. The monitor receives this special monitor record from its monitor input buffer registered on the data chain by calling DosMonRead. The monitor sends a Font Monitor Buffer response to this command (returning the

active code page and font information) to the printer device driver by calling DosMonWrite, to place a special monitor record into its monitor output buffer registered on the code page chain. The printer device driver then returns the active code page and font information to the caller of the Query Active Font IOCTL.

If the spooler, or another monitor, is not registered, the printer device driver will return the appropriate return code to the caller of the Query Active Font IOCTL.

*Verify Font*

When the printer device driver receives the Verify Font IOCTL request, if the spooler or another monitor is registered, the printer device driver sends a Font Monitor Buffer command (in this case, a Verify Font command) to the monitor on the data chain. The monitor receives this special monitor record from its monitor input buffer registered on the data chain by calling DosMonRead. The monitor sends a Font Monitor Buffer response to this command (validating the specified code page and font) to the printer device driver by calling DosMonWrite, to place a special monitor record into its monitor output buffer registered on the code page chain. The printer device driver then returns the return code to the caller of the Verify Font IOCTL.

If the spooler, or another monitor, is not registered, the printer device driver will return the appropriate return code to the caller of the Verify Font IOCTL.

Refer to "Generic IOCTL Interface" on page 6-137 and "Monitor Dispatcher Interface" on page 6-140 for specific interface parameters.

## Timeout Processing

The printer device driver sets a 120-second timer after receiving a write request and sending the first byte to the parallel port. If a hardware interrupt is not generated after 120 seconds, the timer expires and a timeout occurs.

There are two different mechanisms for processing timeouts:

- If infinite retry is enabled, the printer device driver does not terminate the request and indefinitely continues to try to send the data to the printer.
- If infinite retry is disabled, the printer device driver terminates the write request, returning the number of actual bytes sent to the device.

When a monitor is registered, infinite retry is always enabled. The printer device driver sends a status monitor packet through the data chain (index = 1) when a timeout error occurs. This activity alerts all monitors of the error. The spooler currently displays an error message when it encounters a status monitor packet, to alert the user to the hardware problem.

If a hardware interrupt does occur, the printer device driver resets the timer to 120 seconds and sends the next byte to the device. This activity will continue until all data has been sent to the device. Once the printer device driver determines all characters have been sent, it will turn the timer off.

# Printer Device Driver Interfaces

The printer device driver has several interfaces:

1. The request packet interface (referred to as the strategy interface)
2. The generic IOCTL interface
3. The BIOS interrupt 17H interface
4. The hardware interrupt interface
5. The monitor dispatcher notification interface.

## Request Packet Interface

The file system is the primary component which interfaces with the printer device driver. In response to a function call, the file system creates a request packet containing the information required by the printer device driver to process the request. The file system then calls the printer device driver's strategy entry point, with the registers ES:BX containing the address of the request packet.

The strategy interface uses the CALL/FAR return model and must preserve the caller's registers. The strategy entry point routes all IOCTL requests and all open, read, write, close, and status requests to the appropriate internal printer device driver routines to handle the requests. See the description of a request packet in the Device Driver Architecture chapter of this book for the format of a request packet.

### Request Packet Command Summary

The command code field of a request packet contains the function requested of the device driver. The printer device driver is a character device driver and supports all character device driver functions. The command codes supported by the printer device driver are listed in Table 6-7.

Table 6-7. Summary of Printer Device Driver Commands

| Code | Function |
|---|---|
| 0H | Initialize device driver |
| 4H | Read (input) |
| 5H | Nondestructive read no wait |
| 6H | Input status |
| 7H | Input flush |
| 8H | Write (output) |
| 9H | Write with verify |
| AH | Output status |
| BH | Output flush |
| DH | Device open |
| EH | Device close |
| 10H | Generic IOCTL |
| 14H | Deinstall |

**Note:** The status returned in the request packet for requests the printer device driver does not support is 8103H (bad command).

A detailed description of these commands can be found in Chapter 4, "Device Driver Strategy Commands" on page 4-1. "Command Specific Parameters" on page 6-134 describes the printer device driver device-dependent parameters for these commands.

### Request Packet Status Description

On the call to the printer device driver, the status code is set to zero. On return from the device driver, the status code contains the results of the request (for example, done, error). The printer device driver sets the error bit, busy bit, done bit, and error code when necessary. Table 6-8 on page 6-134 lists the bit descriptions for the request packet status.

| Table 6-8. Return status code from printer device driver | | |
|---|---|---|
| **Bit** | **Description** | **Value** |
| 15 | Error Bit | 0 - no error 1 - error |
| 14 | Reserved | 0 |
| 13 | Reserved | 0 |
| 12 | Reserved | 0 |
| 11 | Reserved | 0 |
| 10 | Reserved | 0 |
| 9 | Busy Bit | 0 - not busy 1 - busy |
| 8 | Done Bit | 0 - not done 1 - done |
| 7-0 | Error Code | Hex value |

**Return Codes**

The error codes the printer device driver returns are listed in Table 6-9

| Table 6-9. Error codes returned from printer device driver | |
|---|---|
| **Code** | **Descriptions** |
| 8102H | Device not ready |
| 8103H | Unknown command |
| 8109H | Printer out of paper |
| 810AH | Write fault |
| 810CH | General failure |
| 8111H | Character I/O call interrupted |

The reserved field of the request packet is not used by the printer device driver.

The queue linkage field of the request packet is available for linking request packets together but is not used by the printer device driver.

**Command Specific Parameters**

This section describes the command-specific data parameters required by the printer device driver to respond to the character device driver commands. This data follows the 13-byte request header in the request packet.

*Command: Initialize device driver (command code = 0)*

| Table 6-10 (Page 1 of 2). Command Specific Data for Printer Device Driver | |
|---|---|
| Number of units | BYTE |

| Table 6-10 (Page 2 of 2). Command Specific Data for Printer Device Driver | |
|---|---|
| pointer to DevHlp on entry<br>offset ending address of code seg on exit<br>offset ending address of data seg on exit | DWORD |
| pointer to arguments on entry<br>set to BPB array on exit | DWORD |
| drive number | BYTE |

This request is generated by OS/2 system initialization at INIT time only.

On entry to the printer device driver, the printer device driver saves the 32-bit DevHlp address in its own data segment. When initialization is completed, the printer device driver returns the offset of its code and data segments in the 32-bit DevHlp address parameter. All other parameters are ignored.

The printer device driver returns 0100H (Done, No error) in the request packet status field.

*Command: Read or Write (command code = 4, 8, or 9)*

| Table 6-11. Command Specific Data for Printer Device Driver | |
|---|---|
| Media Descriptor | BYTE |
| Transfer Address | DWORD |
| Byte/Sector Count | WORD |
| Starting Sector Number | DWORD |
| System File Number | WORD |

This request is generated from a file system DosRead or DosWrite API call.

The printer device driver processes the read or write request parameters by using the transfer address as the buffer location to get/put data, and the byte count as the length of that data. The system file number is a unique file number associated with a file handle and is valid for the duration of the DosOpen. All other parameters are ignored.

The printer device driver returns 0100H (Done, No error) in the read request packet status field and zero (0) in the byte count field. If monitors are registered, the printer device device driver returns 0100H (Done, No error) or 810AH (Error, Done, Write fault) in the write request packet status field. If monitors are not registered, the printer device driver returns 0100H (Done, No error), 0102H (Done, Device not ready), 0109H (Done, Printer out of paper), 810AH (Error, Done, Write fault) or 0111H (Done, Character I/O call interrupted) in the write request packet status field.

*Command: Nondestructive read no wait (command code = 5)*

| Table 6-12. Command Specific Data for Printer Device Driver | |
|---|---|
| Returned character | BYTE |

The printer device driver does not use the nondestructive read-no-wait request parameter. The printer device driver returns 0300H (Done, Busy) in the request packet status field.

*Command: Status - Input or Output (command code = 6, 10)*

There are no input status or output status request parameters. If this is an input status request, the printer device driver returns 0100H (Done, No error) in the request packet status field. If an output status request, the printer device driver returns 0100H (Done, No error) or 0300H (Done, Busy) in the request packet status field.

*Command: Flush - Input or Output (command code = 7, 11)*

There are no input flush or output flush request parameters. The printer device driver returns 0100h (Done, No error) in the request packet status field.

*Command: Open or Close a Device (command code = 13, 14)*

| Table 6-13. Command Specific Data for Printer Device Driver | |
|---|---|
| System File Number | WORD |

This request is generated from a file system DosOpen or DosClose API call or from a DosMonOpen or DosMonClose monitor function call. The system file number is a unique number associated with each file handle and is valid for the duration of the DosOpen.

If the request is generated from a DosMonOpen or DosMonClose monitor function call, the request packet status field for the open or close call will be set to 0008h on entry to the printer device driver. This differentiates the request from one generated from a DosOpen or DosClose API call, where the request packet status field for the open or close call is set to a non- 0008h value.

If the request is generated from a DosOpen call, the printer device driver returns 0100H (Done, No error) in the request packet status field. If the request is generated from a DosMonOpen request, the printer device driver returns 0100H (Done, No error) or 810AH (Error, Done, Write fault) in the request packet status field.

*Command: Generic IOCTL - I/O Control for Devices (command code = 16)*

| Table 6-14. Command Specific Data for Printer Device Driver | |
|---|---|
| Category code | BYTE |
| Function code | BYTE |
| 32 bit phys addr of parm buffer | DWORD |
| 32 bit phys addr of data buffer | DWORD |
| System File Number | WORD |

This request is generated from a file system DosDevIOCTL API call. Refer to the "Category 5 Printer Control IOCtl Commands" on page 7-92 for printer device driver-specific parameters and return codes.

Based on the category and function codes, the printer device driver receives data from the caller's buffer located at the specified 32-bit physical address or returns information to the caller's buffer located at another specified 32-bit physical address. The system file number is a unique number associated with each file handle and is valid for the duration of the DosOpen.

If the printer device driver cannot access the caller's parameter buffer or data buffer, it will return 8103H (Error, Done, Invalid command) in the request packet status field.

*Command: Deinstall - Terminate the Device Driver (command code = 20)*

This request is generated by OS/2 system initialization at INIT time only.

There are no deinstall request parameters. The printer device driver will return 0100H (Done, No error) or 8103H (Error, Done, Bad command) if it cannot deinstall (that is, it cannot release its resources).

#### Interrupt 21H Interface

An application running in DOS mode can access the printer device driver through interrupt 21H. The file system creates a request packet from the parameters (passed in registers) and calls the printer device driver at its strategy entry point. This interface has been defined to maintain compatibility with DOS applications. As in the request packet interface, the printer device driver preserves the caller's registers, ES:BX points to the request packet, and the strategy entry point follows the call/far return model. To view the interface from an application's perspective, refer to the DOS Technical Reference manual.

## Generic IOCTL Interface

The printer device driver supports a set of generic IOCTL function calls. Refer to the "Category 5 Printer Control IOCtl Commands" on page 7-92 for a detailed description of each call.

#### Generic IOCTL function calls supported by the Printer Device Driver

The printer device driver supports the following generic IOCTL function calls:

*Category 05H - Printer Control Functions*

**Note:** Supported only by the printer device driver.

| Code | Function |
|---|---|
| **42H** | Set frame control (cpi, lpi) |
| **44H** | Set infinite retry |
| **45H** | Reserved |
| **46H** | Initialize printer |
| **48H** | Activate Font |
| **62H** | Get frame control |
| **64H** | Get infinite retry |
| **66H** | Get printer status |
| **69H** | Query Active Font |
| **6AH** | Verify Font |

*Category 0AH - Character Monitor Control*

**Note:** Supported by the keyboard, mouse and printer device drivers.

**40H** Register

*Category 0BH - General Device Control*

**Note:** Supported by the all character device drivers.

**60H** Query monitor support

**Note:** From the command line, a user can invoke the MODE command to perform the printer control functions: set/get the frame control and set/get infinite retry. See the description of the MODE command in the *OS/2 Version 1.2 Command Reference* for details.

**Character Monitor Control:** The Category 0Ah, Function 40h IOCTL call is supported by the keyboard and mouse device drivers, as well as by the printer device driver. The index field of the data packet received on this call is device-specific. That is, it is defined differently for each device driver. The index field generally indicates on which monitor chain an application wishes to register a monitor.

As previously discussed, the printer device driver creates two monitor chains for each printer device: a data chain and a code page chain. An application issuing this IOCTL function call to a specified printer device sets the index parameter as follows:

- *Index = 1* to register a monitor with the data chain.
- *Index = 2* to register a monitor with the code page chain.

In response to this IOCTL function call, the printer device driver returns the following error codes in the request packet status field:

- 0100H - Completed successfully.
- 8103H - Invalid command if index not 1 or 2.
- 810AH - Write Fault if monitor error returned.

## Interrupt 17H Interface

An application running in DOS mode can access the printer device driver directly by issuing BIOS interrupt 17H print requests. The printer device driver replaces the BIOS interrupt 17H vector address. The printer device driver emulates the interrupt 17H BIOS function in order to synchronizes the BIOS calls to the parallel printer ports with the file system calls to the same ports. The interrupt 17H request is not passed to BIOS for any processing.

The valid interrupt 17H requests include:

- Printing a character
- Initializing a parallel printer port
- Returning a parallel printer port status.

This interface preserves the caller's registers and follows the interrupt/return model. The interrupt 17H interface is described below.

Table 6-15. Register usage for Interrupt 17H

| Register | Description |
|---|---|
| AH = 0 | Print the character in register AL. |
| AH = 1 | Initialize the printer port and return printer port status in AH. |
| AH = 2 | Return printer port status in register AH. |
| DX = 0,1,2 | Printer Port to be used.<br>0 Corresponds to LPT1, 1 to LPT2, and 2 to LPT3. |

### Parallel Port Summary

On the family one machines the printer device driver manipulates the control port to print a character and to initialize the printer. It also returns the status port value on interrupt 17 and generic IOCTL requests. Listed below are the various parallel port addresses and a bit level description of the different ports.

Table 6-16 (Page 1 of 2). Port bit level description

| Address | Description |
|---|---|
| 03BCH | LPT1 - Data port address |

| Table 6-16 (Page 2 of 2). Port bit level description | |
|---|---|
| **Address** | **Description** |
| 03BDH | LPT1 - Status port address |
| 03BEH | LPT1 - Control port address |
| | |
| 0378H | LPT2 - Data port address |
| 0379H | LPT2 - Status port address |
| 037AH | LPT2 - Control port address |
| | |
| 0278H | LPT3 - Data port address |
| 0279H | LPT3 - Status port address |
| 027AH | LPT3 - Control port address |

**Parallel Data Port Description**

The parallel data port is the port where data is put to be sent to a parallel port device (parallel printer). It has no bit level description.

**Parallel Status Port Description**

The parallel status port is described below. The status is the parameter returned on a parallel port status request. Bit 2 has a new meaning on IBM PS/2 machines running ABIOS; it is an interrupt pending bit. When a hardware interrupt is pending, Bit 2 will be equal to 1. The printer device driver masks this bit off (0) so that the status will appear the same to an application running on family one or two machines.

| Table 6-17. Parallel status port bit description | |
|---|---|
| **Bit** | **Description** |
| 7 | Busy bit |
| 6 | Acknowledge bit |
| 5 | Out of paper bit |
| 4 | Selected bit |
| 3 | I/O error bit |
| 2 | Unused (or interrupt pending on IBM PS/2) |
| 1 | Unused |
| 0 | Timeout bit |

**Parallel Control Port Description**

The parallel control port is described below. It is used to send a byte of data to a parallel port device, to instruct the device to run in hardware interrupt mode, and to initialize the parallel port device.

Table 6-18. Parallel control port bit description

| Bit | Description |
|---|---|
| 7 | Unused |
| 6 | Unused |
| 5 | Unused |
| 4 | IRQ enable bit |
| 3 | Select input bit |
| 2 | Initialize port bit |
| 1 | Auto feed bit |
| 0 | Strobe line bit |

## Hardware Interrupt (8259) Interface

When a hardware interrupt is pending on an interrupt level owned by the printer device driver, the hardware interrupt manager calls the printer device driver's hardware interrupt entry point for that interrupt level. The first printer on the family one machines will run on hardware interrupt level 7, the second on hardware interrupt level 5, and the third on a timer. The printer on the family two machines will run only on interrupt level 7. In the future, all parallel printers as well as other devices may share this interrupt level.

Each of the printer device driver's hardware entry points call a printer device driver general interrupt routine. This routine sends the next character to the port, issues an EOI (end of interrupt) and completes a strategy request when the last character has been received by the device.

Since hardware interrupt processing time is faster than the smallest granularity of the timer, printer devices running off the timer run more slowly than printer devices running off hardware interrupts. The function of the routine at the printer device driver's timer entry point is identical to the function of its hardware interrupt routines, with the exception that the device driver returns to timer services instead of the hardware interrupt manager.

The hardware interrupt manager saves all registers before calling the printer device driver interrupt entry points. On return to the hardware interrupt manager, the printer device driver's general interrupt routine clears the carry flag, signalling the interrupt manager that the printer device driver owns the interrupt. If the printer device driver does not own the hardware interrupt, it will set the carry flag.

The printer device driver's hardware interrupt routine follows the call/return far model and does not handle nested interrupts.

## Monitor Dispatcher Interface

The OS/2 printer device driver supports character device monitors. For a description of how that support is provided by the printer device driver, see "Printer Monitors" on page 6-129.

Data sent to each printer device may be monitored before reaching the device by printer monitor applications registered with one of two monitor chains. When a printer monitor is registered, the printer device driver places all open, code page, write (print), and close requests (in the form of monitor records, or packets) into its monitor chains by calling the MonWrite device helper routine.

Each printer monitor registered on the monitor chain can monitor the data destined for the printer. Printer data that has passed through all monitors registered on the monitor chain is placed into a printer device driver monitor chain buffer by the OS/2 monitor dispatcher.

Each monitor chain for a printer device is created by the printer device driver by calling the MonitorCreate device helper routine. For each monitor chain, the printer device driver must provide:

- A device driver *monitor chain buffer* into which filtered printer data, that is, data that has passed through all printer monitors in the monitor chain, is placed by the monitor dispatcher

  **Note:** The length of the printer device driver's monitor chain buffer is 134 BYTES. This length is specified by the printer device driver in the first WORD of the buffer when the MonitorCreate device helper routine is called. The length of printer monitor input and output buffers registered with a printer monitor chain must be greater than or equal to 134 BYTES plus 20 BYTES. See "Buffer Requirements" on page 8-36 for more information.

- The address of a *notification routine*, that is called by the monitor dispatcher when it has placed filtered data into the monitor chain buffer.

  **Note:** Before the monitor dispatcher calls the printer device driver's notification routine, it:

  - Places a data record in the printer device driver's monitor chain buffer starting at the second WORD of the buffer
  - Places the length of the data record (in BYTES) in the first WORD of the printer device driver's monitor chain buffer
  - Sets the register pair ES:SI to point to the printer device driver's monitor chain buffer.

The general monitor packet format for code page processing is described in Table 6-19.

| Table 6-19. Monitor packet for code page processing | |
|---|---|
| Monitor Flags | WORD |
| System File Number | WORD |
| Command Byte | BYTE |
| Reserved, Set to 0 | BYTE |
| Reserved, Set to 0 | BYTE |
| Reserved, Set to 0 | BYTE |
| Return code | WORD |
| Code page | WORD |
| Font ID | WORD |

The general monitor packet format for open, write, close, and status monitor requests is described in Table 6-20.

| Table 6-20. General monitor packet | | |
|---|---|---|
| Monitor flags | | WORD |
| System File Number | | WORD |
| Data to be monitored | (optional) | 128 BYTES |

Data items included in these data structures are defined in the descriptions of the corresponding request packet data structures.

The printer device driver communicates with parallel port monitors using the following monitor protocol.

### Printer Monitor Buffer Command

Each monitor record can be of variable length with a maximum data portion of 128 bytes. However, there must be a word of flags at the beginning of each monitor record. The flags are defined in Figure 6-8.

**Note:** Monitor packets which printer monitors do not understand should be returned to the printer device driver on the monitor chains on which they were received.

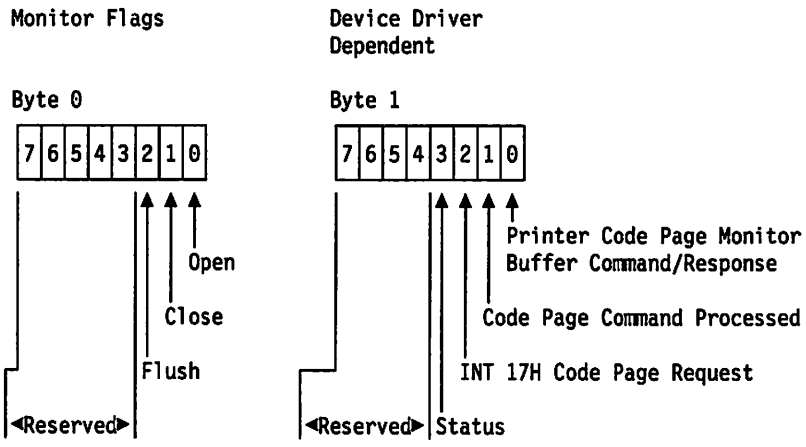



Figure 6-8. Monitor Record Flags

### Monitor Open Packet

When the OPEN bit is set to 1 (byte 0, bit 0), this indicates that the monitor buffer is an open packet from the printer device driver. In this case, the next two bytes are:

| Table 6-21. Monitor Open Packet | |
|---|---|
| System File Number | WORD |

### Monitor Close Packet

When the CLOSE bit is set to 1 (byte 0, bit 1), this indicates that the monitor buffer is a close packet from the printer device driver. In this case, the next two bytes are:

| Table 6-22. Monitor Close Packet | |
|---|---|
| System File Number | WORD |

### Monitor Write Packet

When the CLOSE bit is set to 0 (byte 0, bit 1), and the OPEN bit is set to 0 (byte 0, bit 0), this indicates that the monitor buffer is a write data packet from the printer device driver. In this case, the next bytes are:

| Table 6-23 (Page 1 of 2). Monitor Write Packet | |
|---|---|
| System File Number | WORD |

| Table 6-23 (Page 2 of 2). Monitor Write Packet | |
|---|---|
| Data to be monitored | 128 BYTES |

**Font Monitor Packet**

When the Font Monitor Buffer command/response bit is set to 1 (byte 1, bit 0), this indicates that the monitor buffer is a Font Monitor Buffer command. In this case, the next six bytes are:

| Table 6-24. Monitor Write Packet | |
|---|---|
| System File Number | WORD |
| Command Byte | BYTE |
| Reserved, Set to 0 | BYTE |
| Reserved, Set to 0 | BYTE |
| Reserved, Set to 0 | BYTE |

where:

**Byte 2 & 3** System File Number
**Byte 4** Font Monitor Buffer Command Byte which indicates the type of command or response.
**Byte 5** Reserved, and must be set to zero (0).
**Byte 6 & 7** Reserved, and must be set to zero (0).

When the Code Page Command Processed bit is set to 1 (byte 1, bit 1), this indicates that the Font Monitor Buffer command has been processed by the spooler.

The printer device driver sends Font Monitor Buffer commands to its monitors through the data monitor chain (index = 1). A printer monitor which services the Font Monitor Buffer command will set the Code Page command processed bit and use DosMonWrite to place the Font Monitor Buffer response into the code page monitor chain (index = 2). Data to be printed continues to flow on the data monitor chain (index = 1). The code page monitor chain (index = 2) is used only for the Font Monitor Buffer responses so that the printer device driver does not block a program issuing a code page and font IOCTL request when print data monitor buffers are already queued ahead of the IOCTL request.

When the Interrupt 17H Code Page Request bit is set to 1 (byte 1, bit 2), the Activate Code Page request is issued automatically by the printer device driver on behalf of an application issuing the interrupt 17H.

The valid Font Monitor Buffer commands, along with the remainder of the buffer contents for the responses, are as follows:

**Byte 4 = 01H** *Activate Font*

Command Data, starting at byte 8:

| Table 6-25. Command Data starting at byte 8 | |
|---|---|
| Return Code | WORD |
| Code Page | WORD |
| Font ID | WORD |

where:

**Return Code** A value returned, starting at byte 8, and includes the following values:

- 0000H Successful completion.
- 0002H Code page is not available.
- 0003H No code page function because spooler not started.
- 0004H Font ID is not available(verify).
- 0009H Error caused by switcher error not by input parameters.
- 000AH Error caused by invalid printer name as input.
- 000DH Received code page request when code page switcher not initialized.
- 000FH SFN table full cannot activate another entry.
- 0013H DASD error reading font file.
- 0015H DASD error reading font file definition block.
- 0017H DASD error while writing to temporary spool file.
- 0018H Disk full error while writing to temporary spool file.
- 0019H Spool file handle was bad.

**Code Page** The value of the code page to make the currently active code page.

**0000H** If the Code Page value and Font ID are specified as zero (0), set printer to hardware default code page and font.

**0001H-FFFFH** Valid code page numbers.

**Font ID** The ID value of the font to make currently active.

**0000H** If the Code Page value and Font ID are specified as zero (0), set printer to hardware default code page and font.

If only the Font ID is 0 and the Code Page is a valid non-0, then any font within the specified code page is acceptable.

**0001H-FFFFH** Valid Font ID numbers, font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label and are also entered in the DEVINFO statement for the printer.

The printer device driver passes the Font Monitor Buffer command to the spooler on the data monitor chain (index = 1). The spooler returns the Font Monitor Buffer response to the printer device driver on the code page monitor chain (index = 2).

**Byte 4 = 02H** *Query Active Font*

There is no additional command data. Data returned from this function call includes:

Table 6-26. Data Returned starting at byte 8

| | |
|---|---|
| Return Code | WORD |
| Code Page | WORD |
| Font ID | WORD |

where:

**Return Code** A value returned, starting at byte 8, and includes the following values:

- 0000H Successful completion
- 0003H No code page function because spooler not started
- 0009H Error caused by switcher error not by input parameters
- 000AH Error caused by invalid printer name as input
- 000DH Received code page request when code page switcher not initialized

- 0010H Received request for SFN not in SFN table.

**Code Page** On return, is set to currently active code page.

**0000H** If the Code Page value and Font ID are returned as zero (0), the printer is set to the hardware default code page and font.

**0001H-FFFFH** Valid code page numbers.

**Font ID** On return, is the ID value of the Font which is currently active.

**0000H** If the Code Page value and Font ID are specified as zero (0), the printer is set to the hardware default code page and font.

A Font ID value of zero may indicate the default font of a particular code page.

**0001H-FFFFH** Valid Font ID numbers, font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label and are also entered in the DEVINFO statement for the printer.

The printer device driver passes the Font Monitor Buffer command to the spooler on the data monitor chain (index = 1). The spooler returns the Font Monitor Buffer response to the printer device driver on the code page monitor chain (index = 2).

**Byte 4 = 03H** *Verify Font*

Command Data, starting at byte 8:

| Table 6-27. Command Data starting at byte 8 | |
|---|---|
| Return Code | WORD |
| Code Page | WORD |
| Font ID | WORD |

where:

**Return Code** A value returned, starting at byte 8, and includes the following values:

- 0000H Successful completion
- 0002H Code page is not available
- 0003H No code page function because spooler not started
- 0004H Font ID is not available(verify)
- 000AH Error caused by invalid printer name as input
- 000DH Received code page request when code page switcher not initialized.

**Code Page** The Code Page number to validate.

**0000H-FFFFH** Valid code page numbers.

**Font ID** The Font ID value to validate.

**0000H-FFFFH** Valid Font ID numbers.

The printer device driver passes the Font Monitor Buffer command to the spooler on the data monitor chain (index = 1). The spooler returns the Font Monitor Buffer response to the printer device driver on the code page monitor chain (index = 2).

**Monitor Status Packet**

When the STATUS bit is set to 1 (byte 1, bit 3), this indicates that the monitor buffer is a status packet from the printer device driver. In this case, the next four bytes are:

| Table 6-28. Monitor Status Packet | |
|---|---|
| System File Number | WORD |
| Error code | BYTE |
| Reserved | BYTE |

The printer device driver will send either device not ready (02H), printer out of paper (09H), or write fault (0AH) as the error code.

# Chapter 7. Generic IOCtl Commands

OS/2 device drivers are used by OS/2 to access the I/O hardware. The IOCtl functions provide a method for an application, or subsystem, to send device-specific control commands to a device driver. The IOCtl functions are issued through the DosDevIOCtl API function request. The IOCtl functions are subfunctions of the DosDevIOCtl API function request. Applications should use the DosDevIOCtl function request for OS/2 Applications and the INT 21H IOCtl request for DOS applications.

The category and function fields are determined as follows. Each code is contained in a byte.

## Category Code

| *Category* | *Code* |
|---|---|
| 0... .... | OS/2 Defined |
| 1... .... | User Defined |
| .xxx xxxx | Code |

## Function Code

| *Function* | *Code* |
|---|---|
| 0... .... | Return error if unsupported |
| 1... .... | Ignore if unsupported |
| .0.. .... | Intercepted by OS/2 |
| .1.. .... | Passed to driver |
| ..0. .... | Sends data and commands to device |
| ..1. .... | Queries data and information from device |
| ...x xxxx | Subfunction |

Note that the sends/queries data bit is intended only to regularize the function set. It plays no critical role; some functions may contain both command and query elements. The convention is that such commands are defined as "sends data."

## Generic IOCtl Example

Following is the calling sequence for the DosDevIOCtl call:

```
EXTRN   DosDevIOCtl:Far

PUSH@   OTHER  Data       ;Data Packet
PUSH@   OTHER  ParmList   ;Parameter Packet
PUSH    WORD   Function   ;Function Code
PUSH    WORD   Category   ;Category Code
PUSH    WORD   DevHandle  ;User's Device Driver File Handle

CALL    DosDevIOCtl
```

The DosDevIOCtl call sends the request to the device driver request packet. The device driver receives the request packet, and looks for the Command Code (Command 16 is the Generic IOCtl command) to identify the request.

Note that each device driver can define the structure of the Data Packet and the Parameter Packet, but all device drivers use the same request header.

The list of categories and functions for the GENERIC IOCtl request are summarized in Table 7-1 on page 7-2:

| CAT | FUNCTION | DESCRIPTION |
|---|---|---|
| 01 | | Serial Device Control |
| | 14H | Reserved |
| | 34H | Reserved |
| | 41H | Set baud (bit) rate |
| | 42H | Set line characteristics (stop, parity, data bits) |
| | 44H | Transmit Byte Immediate |
| | 45H | Break off |
| | 46H | Set modem control signals |
| | 47H | Behave as if XOFF received (stop transmit) |
| | 48H | Behave as if XON received (start transmit) |
| | 49H | Reserved |
| | 4BH | Break on |
| | 53H | Set Device Control Block (DCB) parameters |
| | 61H | Return current baud (bit) rate |
| | 62H | Return line characteristics |
| | 64H | Return COM status |
| | 65H | Return transmit data status |
| | 66H | Return modem control output signals |
| | 67H | Return current modem input signals |
| | 68H | Return number of chars in receive queue |
| | 69H | Return number of chars in transmit queue |
| | 6DH | Return COM error |
| | 72H | Return COM event information |
| | 73H | Return Device Control Block (DCB) parameters |
| 02 | | Reserved |
| 03 | 72H | Get Pointer draw address (pointer draw DD) |
| 03 | 70H | Allocate an LDT selector |
| | 71H | Deallocate an LDT selector row. |
| | 73H | Initialize Call Vector Table |
| | 74H | ABIOS pass-through |
| | 75H | Allocate an LDT selector with Offset |
| 04 | | Keyboard Control |
| | 50H | Set code page |
| | 51H | Set input mode (default ASCII) |
| | 52H | Set interim character flags |
| | 53H | Set shift state |
| | 54H | Set typamatic rate and delay |
| | 55H | Notify of change of foreground session |
| | 56H | Set session manager Hot Key |
| | 57H | Set KCB |
| | 58H | Set code page ID |
| | 59H | Set Read Notification |
| | 5AH | Alter Keyboard LEDs |
| | 5BH | Reserved |
| | 5CH | Set NLS & custom code page |
| | 5DH | Create a new logical keyboard |
| | 5EH | Destroy a logical keyboard |
| | 71H | Get input mode |
| | 72H | Get interim character flags |
| | 73H | Get shift state |
| | 74H | Read character data record(s) |
| | 75H | Peek character data record |
| | 76H | Get session manager Hot Key |
| | 77H | Get keyboard type |
| | 78H | Get code page ID |
| | 79H | Translate scan code to ASCII |
| | 7AH | Get Keyboard Hardware ID |
| 05 | | Printer Control |
| | 42H | Set frame control (CPL, LPI) |
| | 44H | Set infinite retry |
| | 45H | Reserved |

| CAT | FUNCTION | DESCRIPTION |
|---|---|---|
| | 46H | Initialize printer |
| | 48H | Activate Font |
| | 62H | Get frame control |
| | 64H | Get infinite retry |
| | 66H | Get printer status |
| | 69H | Query Active Font |
| | 6AH | Verify Font |
| 06 | | Light Pen Control |
| 07 | | Mouse Control |
| | 50H | Allow ptr drawing after screen switch |
| | 51H | Update screen display mode |
| | 52H | Screen switcher call |
| | 53H | Set scaling factors |
| | 54H | Set event mask |
| | 55H | Reserved |
| | 56H | Set pointer shape |
| | 57H | Unmark collision area |
| | 58H | Mark collision area |
| | 59H | Set pointer screen position |
| | 5AH | Set OS/2 mode pointer draw address |
| | 5BH | Set DOS mode pointer draw address |
| | 5CH | Set device status flags |
| | 60H | Get number of buttons |
| | 61H | Get number of mickeys/centimeter |
| | 62H | Get device status flags |
| | 63H | Read event queue |
| | 64H | Get event queue status |
| | 65H | Get event mask |
| | 66H | Get scaling factors |
| | 67H | Get pointer screen position |
| | 68H | Get pointer shape image |
| | 69H | Reserved |
| | 6AH | Return the mouse device driver level/version |
| 08 | | Logical Disk Control |
| | 00H | Lock drive |
| | 01H | Unlock drive |
| | 02H | Redetermine media |
| | 03H | Set logical map |
| | 20H | Block removable |
| | 21H | Get logical map |
| | 22H | Reserved |
| | 43H | Set device parameters |
| | 44H | Write track |
| | 45H | Format and verify track |
| | 5EH | Reserved |
| | 5FH | Reserved |
| | 63H | Get device parameters |
| | 64H | Read track |
| | 65H | Verify track |
| 09 | | Physical Disk Control |
| | 00H | Lock physical drive |
| | 01H | Unlock physical drive |
| | 44H | Physical write track |
| | 63H | Get physical device parameters |
| | 64H | Physical read track |
| | 65H | Physical verify track |
| 10 | | Character Device Monitor Control |
| | 40H | Register |
| 11 | | General Device Control |
| | 01H | Flush input buffer |
| | 02H | Flush output buffer |

| CAT | FUNCTION | DESCRIPTION |
|---|---|---|
| | 41H | Session switch or termination notification |
| | 60H | Query monitor support |
| 12-127 | | Reserved Category Codes |

Table 7-1. General IOCtl Categories and Functions.

# Category 1 ASYNC (RS232-C) Control IOCtl Commands

Wherever null pointer appears, it is the application's responsibility to set up a null pointer for the appropriate packet pointer before calling the device driver. IOCtls may be interpreted differently by future releases if the pointer is not a null pointer. If a NULL POINTER is called for and a null pointer is not received by the device driver, it is considered an invalid parameter or data packet value in this section.

The application cannot assume a given timing relationship between when the IOCtls are executed and when data is received or transmitted by the ASYNC hardware.

Data Carrier Detect (DCD) is the same signal as Receiver Line Signal Detect (RLSD).

The device driver services each communications port (COM1, COM2, ...) independently. IOCtls issued to the device driver for a given port have no effect on any other communications ports that the device driver is servicing.

The following is a summary of Category 1 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 14H | reserved |
| 34H | reserved |
| 41H | Set baud (bit) rate |
| 42H | Set line characteristics (stop, parity, data bits) |
| 44H | Transmit Byte Immediate |
| 45H | Break off |
| 46H | Set modem control signals |
| 47H | Behave as if XOFF received (stop transmit) |
| 48H | Behave as if XON received (start transmit) |
| 49H | reserved |
| 4BH | Break on |
| 53H | Set Device Control Block (DCB) parameters |
| 61H | Return current baud (bit) rate |
| 62H | Return line characteristics |
| 64H | Return COM status |
| 65H | Return transmit data status |
| 66H | Return modem control output signals |
| 67H | Return current modem input signals |
| 68H | Return number of chars in receive queue |
| 69H | Return number of chars in transmit queue |
| 6DH | Return COM error |
| 72H | Return COM event information |
| 73H | Return Device Control Block (DCB) parameters |

# Category 1 –
# Function 41H

Set Baud Rate

## Parameter Packet Format

| Field | Length |
|---|---|
| Bit Rate | WORD |

## Data Packet Format

None. Packet pointer must be NULL.

## Where

***Bit Rate***

The Bit Rate field is a binary integer representing the actual bit rate (bits per second) that the device driver should use to set the bit rate of the COM device. The device driver will set the rate only if the hardware can support the rate within .01% margin of error. The only exceptions are for 110 and 2000 bps, which may have an error of up to .026 and .69%, respectively. In all other cases, if the requested rate cannot be achieved by the hardware within .01% tolerance, the IOCTL will fail with an invalid parameter return code. The recommended bit rate values are:

110
150
300
600
1200
1800
2000
2400
3600
4800
7200
9600
19200 (AT hardware not rated for this speed)

## Returns

If the call is made with invalid Parameter/Data packet values, a general failure error is reported.

## Remarks

If a general failure error is not returned, the device driver will perform the action described in Bit Rate.

An OPEN request packet will not cause the device driver to change the bit rate from its previous value. The initial value is 1200 baud.

The COM device hardware determines which bit rates can be supported on a given system. For more information about the rates that are supported by a specific COM device, see "Special Hardware Considerations."

Set Line Characteristics (stop bits, parity, data bits)

## Parameter Packet Format

| Field | Length |
|---|---|
| Data Bits | BYTE |
| Parity | BYTE |
| Stop Bits | BYTE |

## Data Packet Format

None. Packet pointer must be NULL.

## Where

***Data Bit***

| *Value* | *Meaning* |
|---|---|
| 00H-04H | reserved |
| 05H | 5 data bits |
| 06H | 6 data bits |
| 07H | 7 data bits (initial value) |
| 08H | 8 data bits |
| 09H-FFH | reserved |

***Parity***

| *Value* | *Meaning* |
|---|---|
| 00H | No parity |
| 01H | Odd parity |
| 02H | Even parity (initial value) |
| 03H | Mark parity (parity bit always 1) |
| 04H | Space parity (parity bit always 0) |
| 05H-FFH | reserved |

***Stop Bits***

| *Value* | *Meaning* |
|---|---|
| 00H | 1 stop bit (initial value) |
| 01H | 1.5 stop bits (valid with 5 bit word length only) |
| 02H | 2 stop bits (not valid with 5 bit word length) |
| 03H-FFH | reserved |

## Returns

If the call is made with invalid Parameter/Data packet values, a general failure error is reported and the line characteristics are not changed for any parameters that were valid.

## Remarks

If a general failure error is not returned, the device driver will set the line characteristics as defined.

An OPEN request packet will not cause the device driver to change the line characteristics from its previous values.

If the word length is less than 8 bits, the appropriate high order bits for received data will be 0 when placed in the receive queue by the device driver and when operated on by the device driver (for example, XON/XOFF checking, null stripping). This only applies to data that is received after the command is operated on by the device driver. Data already in the device driver receive queue is not affected in any way by a change in the word length.

## Category 1 – Function 42H

If the word length is less than 8 bits the device driver will not automatically truncate control/transmit data that the application may tell the device driver to operate on or use. No error will be reported by the device driver if transmit or control data given to the device driver has high order bits of non-0 value.

For example, if the device driver is told that the word length is 7 bits (high order bit of all data in receive queue from now on is 0) and the XOFF character is 80H then the device driver will never be able to recognize the XOFF character if automatic transmit flow control is enabled. If the error substitution character is set to 80H by the application with a word length of 7 currently being active, the device driver will still place an 80H in the receive queue. It is the responsibility of the application to maintain consistency between the requested word length for the COM device and the requests that the application makes of the device driver.

Transmit Byte Immediate

## Parameter Packet Format

| Field | Length |
|---|---|
| Character to be Transmitted | BYTE |

## Data Packet Format

Packet pointer must be NULL.

## Returns

If the call is made with an invalid Data Packet value or if there is already another character waiting to be transmitted immediately due to a previous Category 1 Function 44H request that has not been fulfilled then a general failure error is reported and this request is ignored. A transmit immediate request is considered fulfilled when the character is given to the transmit hardware.

## Remarks

If a general failure is not returned, the device driver will immediately transmit the byte contained in the Parameter Packet subject to the following conditions:

1. If there is data currently in the transmit queue being transmitted, or waiting to be transmitted, the character to be transmitted immediately will be placed at the logical front of the device driver transmit queue (not considered in transmit queue) so it is the next character to be given to the transmit hardware. If automatic receive flow control is enabled then a XON or XOFF character may or may not be placed ahead of the character to be transmitted immediately.
2. This request always completes immediately (before the character is actually transmitted) even if the character may not be immediately transmitted for reasons discussed below. If there already is one character waiting to transmit immediately due to a previous request then a general failure error will be returned and the application must make this request again after there is no character waiting to transmit immediately in the device driver transmit queue. Category 1 Function 64H (Return COM status) can be used to determine whether a character is currently waiting to be transmitted immediately.
3. The device driver will not immediately transmit the character waiting to transmit immediately if the device driver is not transmitting characters due to modem control signal output handshaking (see Set Device Control Block (DCB) Category 1 Function 53H Note 3) or if the device driver is currently transmitting a break.
4. If the device driver is not transmitting characters due to automatic transmit or receive flow control (XON/XOFF) being enabled (with the proper set of conditions having happened, see Category 1 Function 53H - Set Device Control Block (DCB)), or due to being asked to behave as if an XOFF character had been received (Category 1 Function 47H) then the device driver will still transmit a character that is waiting to be transmitted immediately due to this request.
   WARNING: An application which requests the device driver to transmit a character immediately when automatic transmit or receive flow control is enabled may cause unexpected results to happen to the communications line flow control protocol.
5. This is generally used to manually send XON and XOFF characters.
6. The character waiting to be transmitted immediately is not considered part of the device driver transmit queue and is not flushed due to a flush request. XON/XOFF characters that are automatically transmitted due to automatic receive flow control may or may not be placed ahead of the character waiting to be transmitted immediately. Applications should not have dependencies on this ordering.
7. If the serial port controller fully supports Extended Hardware Buffering capabilities, and the device driver is set with Extended Hardware Buffering ENABLED, then calling this function will result in temporarily setting the Transmit Buffer Load Count to 1 (to load the transmit immediate byte). If the device driver conditions allow data to be transmitted, then the byte will be sent, and the device driver will then resume operations with the previously prevailing Transmit Buffer Load Count (as determined by the Set Device Control Block Information IOCtl, Category 1, function 53H).

# Category 1 –
# Function 45H

Set Break Off

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| COM Error Word (COMERR) | WORD |

## Where

***COM Error Word***

The device driver returns this information if a general failure error is not reported. See Category 1 Function 6DH, Return COM Error, for COMERR definition. The COM device error information is not cleared by this action.

## Returns

If the call is made with an invalid Parameter Packet value then a general failure error is reported, this function is not performed, and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned then the device driver will stop generating a break signal. It is not considered an error if the device driver is not generating a break signal. The device driver will then resume transmitting characters taking into account all the other reasons why it may or may not transmit characters.

Set Modem Control Signals

## Parameter Packet Format

| Field | Length |
|---|---|
| Modem Control Signals ON Mask | BYTE |
| Modem Control Signals OFF Mask | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| COM Error Word (COMERR) | WORD |

## Where

***Modem Control Signals Value***

The device driver will set the modem control signals as defined in this field. Bit 0 is DTR and bit 1 is RTS. If any other bits are set/reset by the masks then a general failure error results. The OFF mask contains a mask of the bits to turn off. The OFF mask has bits of 0 for the bits to turn off. The ON mask contains a mask of the bits to turn on. The ON mask has bits of 1 for the bits to turn on. If the Parameter Packet shows to turn off and on the same bit, the bit will be turned on.

For example:

| Mask ON | Mask OFF | Meaning |
|---|---|---|
| 01H | FFH | Set DTR |
| 00H | FEH | Clear DTR |
| 02H | FFH | Set RTS |
| 00H | FDH | Clear RTS |
| 03H | FFH | Set DTR and RTS |
| 00H | FCH | Clear DTR and RTS |

If the DTR control mode input handshaking or the RTS control mode input handshaking or toggling on transmit is set then this request is not allowed to try to change the state of the modem control signal(s) that is (are) being used for input handshaking or toggling on transmit. If the request tries to modify a modem control signal that is being used for input handshaking or toggling on transmit then a general failure will result.

***COM Error Word***

The device driver returns this information if a general failure error is not returned to the application. See Category 1, Function 6DH, Return COM Error, for COMERR definition. The COM device error information is not cleared by this action.

At device driver initialization, the device driver will turn OFF DTR and RTS for the COM devices that it owns.

An OPEN request packet, when the COM device is not already open (from a previous open without a close) (first level open) will cause DTR and RTS to be set according to the DTR Control Mode and the RTS Control Mode. See Note 1 of Set Device Control Block (DCB) (Category 1 Function 53H).

**Note:** If the port will not be open any more after processing a close request packet (last level close) DTR and RTS will be turned OFF (by the device driver). This occurs after the transmit hardware has completely transmitted (at the physical RS232 interface) all the data that it has been given to transmit by the device driver and at least 10 additional character times have elapsed.

## Category 1 –
## Function 46H

### Returns

If the call is made with invalid Parameter Packet values then a general failure error is reported, the modem control signals are not changed, and the data packet information returned to the application is undefined.

### Remarks

None

Behave as if XOFF Received (stop transmitting)

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

None. Packet pointer must be NULL.

## Returns

If the call is made with invalid Parameter/Data Packet values, a general failure error is reported and this request is not performed by the device driver.

## Remarks

If a general failure error is not returned by the device driver, this function causes data transmission to be halted by preventing the device driver from sending additional data to the transmit hardware.

If automatic transmit flow control is enabled then this request causes the device driver to behave exactly as if it received the XOFF character. Transmission can be resumed (due to being stopped from this request) when an XON is received by the device driver, when a Category 1 Function 48H (Behave as if XON received) request is received, or when the device driver is told to disable automatic transmit flow control and the previous state was that automatic transmit flow control was enabled.

If automatic transmit flow control is disabled then a Category 1 Function 48H (Behave as if XON received) request is required for transmission to be resumed (due to being stopped from this request). If after this request is received, the device driver is told to enable automatic transmit flow control then transmission is still disabled but can be re-enabled (due to being stopped from this request) by any of the scenarios discussed in the automatic transmit flow control being enabled scenario.

**Note:** There still may be other reasons why transmission may be disabled. (See Return COM Status, Category 1 Function 64H.)

Behave as if XON Received (start transmitting)

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

None. Packet pointer must be NULL.

## Returns

If the call is made with invalid Parameter/Data Packet values, a general failure error is reported and this request is not performed by the device driver.

## Remarks

If a general failure error is not returned by the device driver, this function allows data transmission to be resumed by the device driver if data transmission is halted due to a Category 1 function 47H (Behave as if XOFF received) request or due to an XOFF character being received while the device driver is in automatic transmit flow control mode.

**Note:** There still may be other reasons why transmission may be disabled; so transmission may not be resumed. (See Return COM Status, Category 1 Function 64H.)

Set Break On

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| COM Error Word (COMERR) | WORD |

## Where

***COM Error Word***

The device driver returns this information if a general failure error is not reported. See Category 1 Function 6DH, Return COM Error, for COMERR definition. The COM device error information is not cleared by this action.

A CLOSE request packet, when after processing this close request the port will not be open any more (from another open without a close) will cause break to be turned off.

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported, this function is not performed and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned, the device driver will perform the following action:

The device driver will generate the break signal immediately. It is not considered an error if the device driver is already generating a break signal. The device driver will not wait for the transmit hardware to become empty. Note that more data will not be given to the transmit hardware until the break is turned off. The break signal will always be transmitted, regardless of whether the device driver is or is not transmitting characters due to other reasons.

# Category 1 – Function 53H

Set Device Control Block (DCB)

## Parameter Packet Format

| Field | Length |
|---|---|
| Write Timeout | WORD |
| Read Timeout | WORD |
| Flags1 | BYTE |
| Flags2 | BYTE |
| Flags3 | BYTE |
| Error Replacement Character | BYTE |
| Break Replacement Character | BYTE |
| XON Character | BYTE |
| XOFF Character | BYTE |

## Data Packet Format

None. Packet pointer must be NULL.

## Where

***Write Timeout***

specifies the time period used for write timeout processing. The value is in .01 second units based on 0, where 0 equals .01 seconds. (Refer to Note 8: Write Timeout later in this chapter.)

***Read Timeout***

specifies the time period used for read timeout processing. The value is in .01 second units based on 0, where 0 equals .01 seconds. (Refer to Note 9: Read Timeout later in this chapter.)

***Flags1:***

| Note | Bit | Meaning |
|---|---|---|
| 1 | 0-1 | DTR Control Mode<br>*Bit 1* *Bit 0*<br>0 0 Disable<br>0 1 Enable<br>1 0 Input handshaking<br>1 1 Invalid input resulting in general failure |
| | 2 | reserved (set to 0) |
| 3 | 3 | Enable output handshaking using CTS |
| 3 | 4 | Enable output handshaking using DSR |
| 3 | 5 | Enable output handshaking using DCD |
| 4 | 6 | Enable input sensitivity using DSR |
| | 7 | reserved (set to 0) |

***Flags2:***

| Note | Bit | Meaning |
|---|---|---|
| 2 | 0 | Enable automatic transmit flow control (XON/XOFF) |
| 2 | 1 | Enable automatic receive flow control (XON/XOFF) |
| 5 | 2 | Enable error replacement character |
| 6 | 3 | Enable null stripping (remove null bytes) |
| 7 | 4 | Enable break replacement character |

| Note | Bit | Meaning |
|---|---|---|
| 2 | 5 | Automatic Receive Flow Control<br>0 = Normal<br>1 = Full-Duplex |
| 1 | 6-7 | RTS Control Mode<br>*Bit 7* *Bit 6*<br>0 0 Disable<br>0 1 Enable<br>1 0 Input handshaking<br>1 1 Toggling on transmit |

***Flags3:***

| Note | Bit | Meaning |
|---|---|---|
| 8 | 0 | Enable write infinite time out processing |
| 9 | 1-2 | Read timeout processing<br>*Bit 2* *Bit 1*<br>0 0 Invalid input resulting in general failure<br>0 1 Normal read timeout processing<br>1 0 Wait for something, read timeout processing<br>1 1 No wait, read timeout processing |
| 10 | 3-4 | Extended Hardware Buffering<br>*Bit 4* *Bit 3*<br>0 0 Not Supported; ignored. No change to FIFO state.<br>0 1 Extended Hardware Buffering DISABLED<br>1 0 Extended Hardware Buffering ENABLED<br>1 1 Automatic Protocol Override |
| 11 | 5-6 | Receive Trigger Level<br>*Bit 6* *Bit 5*<br>0 0 1 character<br>0 1 4 characters<br>1 0 8 characters<br>1 1 14 characters |
| 12 | 7 | Transmit Buffer Load Count<br>0 = 1 character<br>1 = 16 characters |

***Error Replacement Character***
any byte value in the range 00H to FFH. Refer to Note 5: Error Replacement Character later in this chapter.

***Break Replacement Character***
any byte value in the range 00H to FFH. Refer to Note 7: Break Replacement Character later in this chapter.

***XON Character***
any byte value in the range 00H to FFH. Refer to Note 2: Automatic Flow Control later in this chapter.

***XOFF Character***
any byte value in the range 00H to FFH. Refer to Note 2: Automatic Flow Control later in this chapter.

# Category 1 – Function 53H

## Returns

If the call is made with invalid Parameter/Data Packet values, a general failure error is reported and none of the Device Control Block (DCB) characteristics of the device driver for this COM device are changed.

## Remarks

The general Device Control Block (DCB) parameter access functions (53H and 73H) are used for:

- Automatic transmit flow control (start/stop transmit when XON/XOFF character received)
- Automatic receive flow control (transmit XON/XOFF when receive buffer fills/empties)
- Determine XON/XOFF characters
- DTR control mode (enable/disable/input handshaking)
- RTS control mode (enable/disable/input handshaking/toggling on transmit)
- Output handshaking using CTS/DSR/DCD (control signal determines when to transmit)
- Input sensitivity using DSR (reception of data controlled by DSR)
- Error replacement character and processing
- Break replacement character and processing
- Null stripping
- Receive/transmit time out processing.

**Note:** To maintain upward compatibility, the application should do a Return Device Control Block (DCB) information before the Set function is used. This will allow the reserved (set to 0) bits to be set correctly in a future release of the device driver when these bit positions may take on a real meaning. By doing the return first the application can maintain the state of the device driver for a mode that the application is not aware of.

**Notes:**

1. Control of DTR and RTS

   The device driver allows the caller to automatically control the setting of Data Terminal Ready (DTR) and Request To Send (RTS) in many different ways through the RTS Control Mode and the DTR Control Mode settings of the Set Device Control Block IOCtl. The application can also request manual control over these modem control signals. The ways these signals are controllable are as follows:

   **Set RTS Control Mode to Toggling on Transmit.**

   If bits 7,6 of Flags 2 are set to 1,1 then the device driver is in this automatic control mode of RTS. When the device driver is initialized, the RTS Control Mode is Enable; so initially the device driver is not in this automatic control mode of RTS.

   **Note:** This mode of operation of the device driver should only be enabled when the system is attached to devices which will not present data to the system receive hardware when RTS is on.

   In this mode, the device driver will:

   - Always turn on RTS if a break is being transmitted.
   - Once data is in the transmit hardware buffer, the device driver will not turn RTS off until the transmit hardware has emptied its buffers.
   - Turn on RTS (if not already on) if there is data in the device driver transmit queue OR if there is an outstanding WRITE request packet and:
     - The device driver is allowed to transmit even if automatic transmit/receive flow control (XON/XOFF) is enabled. The device driver still needs to turn on RTS momentarily to transmit a character immediate if not normally allowed to transmit due to automatic transmit/receive flow control (XON/XOFF).
     - The device driver is allowed to transmit because it was not told to behave as if an XOFF had been received (Category 1 Function 47H). The device driver will still need to turn on RTS momentarily to transmit a character immediate if not normally transmitting due to XOFF flow control considerations. The device driver will still need to turn on RTS momentarily to transmit an XON or XOFF due to automatic

receive flow control if not normally transmitting due to XOFF flow control considerations.

- Turn off RTS (if not already off) if either of the following conditions are true:
  - No more data in the device driver transmit queue (and no more data in WRITE requests in progress), no queued WRITE requests, and the transmit hardware has physically transmitted (at the physical RS232 interface) all the data that it has been given.
  - The device driver is not allowed to transmit due to transmit/receive flow control (XON/XOFF) being enabled or due to being asked to behave as if an XOFF had been received (Category 1 Function 47H). The device driver still needs to turn on RTS to transmit a character immediate or XON/XOFF due to automatic receive flow control (XON/XOFF). RTS is never turned off until the transmit hardware has physically transmitted (at the physical RS232 interface) all the data that it has been given.
- When this function is enabled, the device driver will control RTS appropriately, as determined by the above description.
- If this function is disabled (by choosing a new RTS Control Mode), then RTS will be controlled appropriately by the new RTS Control Mode that is inherently chosen when this RTS Control Mode is disabled.
- The device driver will NOT examine any other modem control signals before it turns RTS off or on.

An OPEN request packet will not cause the device driver to change the RTS Control Mode that the device driver is in. The device driver will maintain the state of this mode of operation across OPEN request packets.

When the device driver is in the RTS Control Mode toggling on transmit, then the device driver will not allow the application to control RTS by the Set Modem Control Signals (Category 1 Function 46H).

**Set DTR and/or RTS Control Mode to Input Handshaking.**

Setting bits 1,0 of Flags 1 to 1,0 sets the DTR Control Mode to Input Handshaking. Setting bits 7,6 of Flags 2 to 1,0 sets the RTS control mode to Input Handshaking. When the device driver is initialized, the RTS and DTR Control Mode is Enable; so initially the device driver is not in this automatic control mode of RTS and DTR.

**Note:** This mode of operation of the device driver should only be set when there is the possibility of a device driver RECEIVE QUEUE overrun and the system is attached to data terminal equipment which will stop transmitting data when the appropriate modem control signals are turned off, due to the cabling and the data terminal equipment characteristics.

Because Input Handshaking mode can be set for either RTS or DTR or both, the DTR and RTS Control Modes are processed independently.

In Input Handshaking mode the device driver will:

- Turn the appropriate modem control signal(s) ON when the device driver receive queue is less than about half full.
- Turn the appropriate modem control signal(s) OFF when the device driver receive queue gets close to full.
- When this mode is first set, the device driver will not monitor the value of the appropriate modem control signal(s) (DTR or RTS) when the queue size is between approximately half full and almost full.
- When this function is enabled, the device driver will determine the correct value of the modem control signal(s) and control them accordingly.
- If this function is disabled (by choosing a new RTS and/or DTR Control Mode), RTS and/or DTR will be controlled appropriately by the new RTS and/or DTR Control Mode that is inherently chosen when this RTS and/or DTR Control Mode is disabled.

- The device driver will NOT examine any other modem control signals before controlling DTR or RTS due to this mode.

An OPEN request packet will not cause the device driver to change the RTS and DTR Control Modes that the device driver is in. The device driver will maintain the state of these modes of operation across OPEN request packets.

When the device driver is in the RTS Control Mode Input Handshaking then the device driver will not allow the application to control RTS through Set Modem Control Signals (Category 1 Function 46H). When the device driver is in the DTR Control Mode Input Handshaking, then the device driver will not allow the application to control DTR through Set Modem Control Signals (Category 1 Function 46H).

**Set DTR and/or RTS Control Mode to Enable or Disable.**

OPEN processing

- Setting bits 1,0 of Flags 1 to 0,0 sets the DTR Control Mode to Disable.
- Setting bits 1,0 of Flags 1 to 0,1 sets the DTR Control Mode to Enable.
- Setting bits 7,6 of Flags 2 to 0,0 sets the RTS control mode to Disable.
- Setting bits 7,6 of Flags 2 to 0,1 sets the RTS control mode to Enable.

When the device driver is initialized, the RTS and DTR Control Mode is Enable, but the value of the modem control signals is OFF until the port gets an OPEN request packet.

An OPEN request packet will not cause the device driver to change the RTS and DTR Control Modes that the device driver is in. The device driver will maintain the state of these modes of operation across OPEN request packets.

Because Enable or Disable modes can be set for either RTS or DTR or both, the DTR and RTS Control Modes are processed independently. The following discussion covers what happens to RTS. The same discussion also applies to DTR if the DTR Control Mode is set as described in the RTS discussion.

If the RTS Control Mode is Disable, when the device driver receives an OPEN request packet and the device is not already open (from a previous open without a close - First Level Open), the RTS modem control signal will be kept (turned) OFF during the OPEN processing. If the RTS Control Mode is Enable then when the device driver receives an OPEN request packet and the device is not already open (from a previous open without a close - First Level Open), the RTS modem control signal will be turned ON during the OPEN processing.

If the RTS Control Mode is set to Disable and the previous mode was not Disable, the RTS modem control signal is turned OFF. If the RTS Control Mode is set to Disable and the previous mode was also Disable, this IOCtl has no effect on the RTS modem control signal.

If the RTS Control Mode is set to Enable and the previous mode was not Enable, the RTS modem control signal is turned ON. If the RTS Control Mode is set to Enable and the previous mode was also Enable, this IOCtl has no effect on the RTS modem control signal.

The following summarizes the previous discussion:

```
DTR/RTS:
(IH= Input Handshaking)
FROM(1) TO(1)  EFFECT(2)
------- ------- -------
Disable Disable none
Disable Enable  turn ON
Disable  IH     auto(3)

Enable  Disable turn OFF
Enable  Enable  none
Enable   IH     auto(3)

 IH     Disable turn OFF
 IH     Enable  turn ON
 IH      IH     auto(3)

RTS ONLY:
(toggle = toggle on transmit)

Disable toggle  auto(4)
Enable  toggle  auto(4)
 IH     toggle  auto(4)

toggle  Disable turn OFF
toggle  Enable  turn ON
toggle   IH     auto(3)
toggle  toggle  auto(4)
```

(1) - From or To Control Mode.

(2) - Effect on the modem control signal.

(3) - Modem control signal controlled automatically (See the section on Input Handshaking).

(4) - Modem control signal controlled automatically (See the section on Toggling on Transmit).

Because the initial Control Mode of the device driver is Enable for RTS and DTR, both modem control signals will be turned ON when the port is first opened.

If the device driver receives an OPEN request packet and the device is already open, the device driver does not alter the value of the RTS and DTR modem control signals, regardless of the Control Mode.

Application control of DTR and RTS.

The application can explicitly turn DTR or RTS ON or OFF (independently) with the Set Modem Control Signals IOCtl (Category 1 Function 46H).

If the Control Mode of RTS is not Enable or Disable, the application may not control RTS with the Set Modem Control Signals IOCtl because the device driver is controlling the signal automatically (toggling on transmit or input handshaking). If the Control Mode of DTR is not Enable or Disable, the application may not control DTR with the Set Modem Control Signals IOCtl because the device drive is controlling the signal automatically (input handshaking).

CLOSE processing.

If the device driver receives a CLOSE request packet and the COM device will still be open (from another open without a close) then the device driver will not change the values of DTR or RTS.

If the device driver receives a CLOSE request packet, when after processing this close request the port will not be open any more (from another open without a close - Last Level Close) then, at the end of the CLOSE processing, RTS and DTR will be turned OFF by the device driver; after waiting the appropriate amount of time. (See description of

CLOSE processing on page 6-19 and IOCtl Set Modem Control Signals, Category 1 Function 46H).

2. Automatic Flow Control. XON/XOFF Characters

   If bit 0 of Flags 2 is set, the device driver is enabled for automatic transmit flow control. If bit 1 is set, the device driver is enabled for automatic receive flow control.

   When the device driver is initialized these bits are reset, so initially the device driver is not enabled for automatic transmit or receive flow control.

   An OPEN request packet will not cause the device driver to change the enabling or disabling state of automatic transmit/receive flow control.

   An OPEN request packet, when the COM device is not already open (from a previous open without a close - First Level Open), will cause the device driver to believe it has not received an XOFF if automatic transmit flow control is enabled and will cause the device driver to believe it has not transmitted an XOFF if automatic receive flow control is enabled.

   **Automatic Transmit Flow Control (XON/XOFF)**

   In the discussion that follows it is stated in places that the device driver will transmit XON or XOFF. There are reasons why the device driver may not be able to transmit an XON or XOFF (transmitting break, invalid output handshaking on modem control signals). It is also stated that the device driver will resume transmitting data. There are also other potential reasons (not related to automatic transmit/receive flow control) why that may not be possible. See Return COM Status IOCtl (Category 1 Function 64H).

   When XON and XOFF flow control during transmission is enabled, the device driver will stop sending data to the transmit hardware when an XOFF is received, and resume sending data to the transmit hardware when an XON is received. (Reminder: Transmission may not be possible due to other reasons). The device driver will still transmit characters due to the transmit immediate request IOCtl (Category 1 Function 44H). The device driver will still transmit XON and XOFF due to automatic receive flow control.

   When the device driver is in this mode, it will not pass received XON and XOFF characters to the application. Instead, the device driver will act upon receiving those characters and throw them away.

   The device driver may transmit additional characters before it recognizes an XOFF character which it has not read but which may be in the receive buffer of the hardware. The extent of this scenario will be minimized, but the combined transmit/receive Advanced BIOS request block will still be used on systems that support Advanced BIOS. If the system is relatively slow in responding to interrupts compared to the current baud rate, receive buffer overruns may not be occurring, but the device driver may be apparently slow in responding to an XOFF character.

   If automatic transmit flow control is disabled (after currently enabled) and transmission was not occurring due to an XOFF being received or IOCtl Category 1 Function 47H (behave as if XOFF received) being requested, then transmission will be resumed. (Reminder: transmission may not be resumed for other reasons.)

   Reference Automatic Transmit Flow Control (XON/XOFF) on page 6-12 for the effect of OPEN request packets on this mode of the device driver. It is the application's responsibility not to fully close the port in a way that will cause the device driver to illegally transmit characters when the port is re-opened after being fully closed (First Level Open).

   Output handshaking using modem control signals is another way that the device driver can be told to stop transmitting. See Note 3.

   **Automatic Receive Flow Control (XON/XOFF)**

   When XON and XOFF flow control during receive is enabled, the device driver will transmit an XOFF when its receive queue gets close to full, and an XON when its receive queue is about half full.

When the COM device driver is in the NORMAL mode of Automatic Receive Flow Control after the XOFF is sent, the COM device driver will send no characters until it sends an XON (due to the amount of data in its receive queue being reduced). This is to accommodate those systems that interpret the first character received after an XOFF as an XON, regardless of what the character actually is. The device driver will still transmit characters due to the transmit immediate request (Category 1 Function 44H).

When the COM device driver is in the FULL-DUPLEX mode of Automatic Receive Flow Control, after the XOFF is sent, the COM device driver will continue to send characters even though the receive queue remains nearly full (provided that all other conditions permit data to be sent). This mode of the device driver is set using bit 5 of Flags2.

The device driver will not be able to transmit an XOFF or XON if it is transmitting a break. The device driver will not be able to automatically transmit an XOFF or XON if it is enabled for output handshaking, with certain modem control signals, and those modem control signals are not ON. This could cause a deadlock if the device driver wishes to transmit an XON and it cannot. The device driver will remember that it wanted to transmit an XOFF or XON and will still do so when transmit conditions permit; assuming the receive queue conditions still warrant it.

The device driver will not monitor characters being transmitted by WRITE request packets to see if any of them are XON or XOFF. The device driver will also not monitor characters transmitted immediately (Category 1 Function 44H). For example, the device driver will not stop transmitting characters if the application causes the device driver to explicitly transmit an XOFF.

If automatic receive flow control is enabled (after currently disabled), the device driver will immediately check the receive queue level to see if an XOFF needs to be transmitted. An XON is never transmitted immediately due to this function being enabled. The device driver will only automatically transmit an XON character after it has automatically transmitted an XOFF character.

If automatic receive flow control is disabled (after currently enabled), and transmission was not occurring due to an XOFF being automatically transmitted (only in the NORMAL mode of Automatic Receive Flow Control), the device driver will transmit an XON and transmission will be resumed if possible. (Reminder: transmission may not be taking place for other reasons.)

If Normal Automatic Receive Flow Control is currently enabled, and transmission is not occurring due to an XOFF being automatically transmitted, then when the device driver is called to enable Full-Duplex Automatic Receive Flow Control, the device driver will immediately begin sending data regardless of the state of the receive queue. An XON character will be sent only when the receive queue becomes half full.

If the device driver previously automatically transmitted an XOFF, and a CLOSE request packet is received, when after processing this close request the port will not be open any more (from another open without a close), the device driver will automatically transmit an XON if possible.

Reference Automatic Receive Flow Control (XON/XOFF) on page 6-12 for the effect of OPEN request packets on this mode of the device driver. It is the application's responsibility not to fully close the port in a way that will cause the device driver to illegally transmit characters or the communications link to be in a deadlock state when the port is re-opened after being fully closed (First Level Open).

Input handshaking using modem control signals is another way that the device driver can tell another device to stop transmitting. See Note 1.

**XON and XOFF CHARACTERS**

The value of these bytes in the device control block determine the value of the XON and XOFF character that is used for automatic transmit and receive flow control.

When the XON and XOFF characters are referred to in the Category 1 IOCtl section, the reference is to the value of the XON and XOFF character as determined by this IOCtl.

When the device driver is first initialized, the XON character is 11H and the XOFF character is 13H. An OPEN request packet, when the COM device is not already open, (from a previous open without a close - First Level Open), will cause the XON character to be set to 11H and the XOFF character to be set to 13H.

If the XON and XOFF characters are set equal with this IOCtl, the results are UNDEFINED.

3. Output Handshaking Using CTS, DSR, DCD

   Bits 3, 4, and 5 of Flags 1 control output handshaking using CTS, DSR, and DCD respectively. If the bit is set, output handshaking for the appropriate modem control signal is enabled.

   Output Handshaking mode can be enabled for any combination of CTS, DSR, or DCD because bit 3, 4, and 5 of Flags 1 are processed independently.

   When the device driver is initialized, bits 3 and 4 of Flags1 are set and bit 5 of Flags1 is reset; Therefore, so initially the device driver is enabled for output handshaking using CTS and DSR but disabled for output handshaking using DCD.

   Except for attachment to special devices and/or special cables, output handshaking using DCD should not be enabled.

   Disabling output handshaking using CTS and/or DSR will cause unexpected results when the system is attached to data terminal devices or data communications devices that toggle CTS and/or DSR in order to control the ability of the system to transmit data.

   If the device driver is enabled for this mode of operation, the device driver will be effected in the following manner if the appropriate modem signal(s) are OFF:

   - The device driver will be unable to move data from the device driver transmit queue to the transmit hardware.
   - The device driver will be unable to transmit a character immediately (Category 1 Function 44H) so the character is remembered by the device driver.
   - The device driver will be unable to automatically transmit XONs and XOFFs. The device driver may wish to transmit XONs and XOFFs as a result of automatic receive flow control being enabled.
   - The device driver will still generate a break immediately if requested.
   - The value of CTS, DSR, and DCD do not affect how the device driver controls RTS and DTR.

   An OPEN request packet will not cause the device driver to change the value of bits 3, 4, and 5 of Flags 1. The device driver will maintain the state of this mode of operation across OPEN request packets.

   On devices with a transmit holding register and transmit shift register, the transmit holding register will always be given another character to transmit when it empties (even though a character may still be in the transmit shift register), unless the device driver determines that it is not allowed to transmit any more.

   The device driver will always attempt to detect a change in the modem status signals (CTS, DSR, DCD) before transmitting more data. This requires bypassing the natural priority of the current ASYNC hardware and requires additional complexity in the Advanced BIOS implementation. This feature prevents a temporary elongation of interrupt latency from not allowing the device driver to recognize a change in a modem control signal. (The modem control signal change happened many character times before the transmit hardware is requesting that another character be given to it).

4. Input Sensitivity Using DSR

   Bit 6 of Flags 1 controls input sensitivity using DSR. If the bit is set, input sensitivity using DSR is enabled.

   When the device driver is initialized, bit 6 of Flags 1 is set; so initially the device driver is enabled for input sensitivity using DSR.

   **Note:** Disabling input sensitivity using DSR will cause unexpected results when the system is attached to data terminal devices or data communications devices that toggle DSR when they generate spurious data that they do not wish the system to receive.

If the device driver is enabled for this mode of operation, the device driver will throw away all data input from the receive hardware if DSR is off.

If the device driver processes a change in the DSR modem control signal from ON to OFF or OFF to ON at the same time that it inputs a character from the receive hardware, the device driver will still accept that last character(s). This will prevent a temporary elongation of interrupt latency from causing the device driver to discard a valid character(s). However this could cause the device driver to attempt to process invalid data for one service period of the receive hardware. This requires that the change in the modem control signal gets processed before the device driver attempts to receive data from the receive hardware (See Note 3); or that the received data is saved until a change in modem status (during the same hardware service instance) can be determined.

An OPEN request packet will not cause the device driver to change the value of bit 6 of Flags 1. The device driver will maintain the state of this mode of operation across OPEN request packets.

5. Error Replacement Character

   Bit 2 in Flags 2 controls the enabling of error replacement character processing. If the bit is set, the error replacement character processing is enabled.

   When the device driver is initialized this bit is reset, so initially the device driver is not enabled for the error replacement character. An OPEN request packet, when the COM device is not already open (from a previous open without a close - First Level Open) will cause this bit to be reset, disabling error replacement character processing.

   When the device driver is initialized, the error replacement character is 00H. An OPEN request packet, when the COM device is not already open (from a previous open without a close) will cause the error replacement character to be set back to a 00H.

   If error replacement character processing is disabled, the following applies:

   - If a parity or framing error occurs and if the character with the error is available in the receive hardware buffer, it is placed in the device driver receive queue.
   - If a hardware or receive queue overrun occurs then nothing special is placed in the receive queue to designate an overrun.

   If error replacement character processing is enabled, the following applies:

   - If a parity or framing error occurs, the error replacement character is placed in the device driver receive queue. (The character in the receive hardware buffer, if it was available, is not placed in the receive queue.)
   - If a hardware buffer overrun occurs, the error replacement character is placed in the device driver receive queue to mark the position that a receive overrun occurred. If valid data is in the receive hardware buffer, it is still placed in the device driver receive queue. The processing of the valid data takes place after the hardware buffer overrun condition is recorded in the device driver receive queue.
   - If a device driver receive queue overrun occurs, the last character in the receive queue is replaced with the error replacement character. This allows the application to know the position of where the error occurred. This error replacement (if enabled) will always take precedence over an error replacement or break replacement event that occurred at the same character time.

   Regardless of whether error replacement character processing is enabled, null stripping and checking for XON/XOFF characters will not occur if the character had an error.

   This IOCtl can be used to change the error replacement character by changing the byte representing the error replacement character.

6. Null Stripping

   Bit 3 in Flags 2 controls the enabling of null stripping processing. If the bit is set, null stripping processing is enabled.

   When the device driver is initialized this bit is reset, so initially the device driver is not enabled for null stripping. An OPEN request packet, when the COM device is not already open (from a

previous open without a close - First Level Open) will cause this bit to be reset; disabling null stripping.

If the device driver is enabled for null stripping when characters are read in from the receive hardware any (non-error or non-break) characters with a value of 00H are thrown away, are not checked even if the XON or XOFF character has been set to 00H, and are not placed in the device driver receive queue.

**Note:** Simultaneously setting the XON or XOFF character to 00H, enabling automatic transmit flow control, and enabling null stripping may cause unexpected results but is not considered an error condition by the device driver error checking logic.

7. Break Replacement Character

   Bit 4 in Flags 2 controls the enabling of break replacement character processing. If the bit is set, the break replacement character processing is enabled.

   When the device driver is initialized this bit is reset, so initially the device driver is not enabled for the break replacement character. An OPEN request packet, when the COM device is not already open (from a previous open without a close - First Level Open) will cause this bit to be reset, disabling break replacement character processing.

   When the device driver is initialized, the break replacement character is 00H. An OPEN request packet, when the COM device is not already open (from a previous open without a close - First Level Open) will cause the break replacement character to be reset back to a 00H.

   If break replacement character processing is disabled, the device driver will not place any character in the device driver receive queue when it detects a break condition on the line. A detected break condition has no effect on XON/XOFF detection.

   If break replacement character processing is enabled, when the device driver detects a break condition, it will place the break replacement character in the device driver receive queue.

   If break replacement character processing is enabled, null stripping and checking for XON/XOFF characters will not operate on the break replacement character.

   This IOCtl can be used to change the break replacement character by changing the byte representing the break replacement character.

   If a parity or framing error is generated due to the reception of a break, error replacement processing is not done (except for the overrun condition), break replacement processing is done.

8. Write Timeout

   Bit 0 in Flags3 controls the characteristics of Write timeout processing. If the bit is 0, Write timeout processing uses the value in the Write timeout word in the device control block. If the bit is 1, Write timeout processing is infinite time out.

   The value in the Write timeout word is in .01 second units (based on 0 where, 0 = .01 seconds). The device driver is considered doing NORMAL Write timeout processing when the Write timeout word is used for Write timeout processing.

   During NORMAL Write timeout processing, if the device driver does not give ANY data to the transmit hardware (from the transmit queue) within the period of time specified by the Write timeout word (due to some reason that prevents the device driver from transmitting data), the request will be completed. The accuracy of the timeout period MAY be determined by the request packet being blocked in the device driver and how long it takes for the thread to be dispatched once it is made ready by the timeout period expiring; OR the accuracy of the timeout period MAY be determined by the accuracy of the device driver timer tick processing. If any data had been given to the transmit hardware in that timeout period, the specified period of time will be waited for again, to see if any more data had been transmitted.

   If the timeout period is changed by this IOCtl (or to infinite time out), the new time may take effect immediately or may take effect after the next character is written.

   During Write infinite timeout processing, the request will not complete until all the data from the request has been given to the transmit hardware. The thread of the Write request will not return to the system until the request completes. The device driver will check to see if an IOCtl has

changed the Write timeout processing characteristics at least every minute. This could occur almost immediately (accuracy MAY be determined by the request packet being blocked and/or by device driver timer ticks). This will insure that the device driver periodically checks to see if Write INFINITE timeout processing had been changed to NORMAL Write timeout processing.

As discussed above, the Write timeout characteristics can be changed in the middle of the processing of a Write request and the new timeout attribute is guaranteed to eventually take effect.

When the device driver initializes, NORMAL Write timeout processing is in effect.

When the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close - First Level Open), the value in the Write timeout word is set to 1 minute. The current Write timeout processing characteristics (NORMAL or INFINITE) is not affected.

9. Read Timeout

Bits 2, 1 of flags 3, control the Read timeout processing characteristics of the device driver. The three possible types of Read timeout processing are:

- Normal (Bits 2,1= 0,1)
- Wait For Something (Bits 2,1= 1,0)
- NO WAIT (Bits 2,1= 1,1)

The value in the Read timeout word is in .01 second units (based on 0, where 0 = .01 seconds). The device driver uses the value in the Read timeout word for NORMAL and WAIT FOR SOMETHING Read timeout processing. The accuracy of the time interval MAY be determined by the request being blocked in the device driver and/or by device driver timer ticks.

If the device driver is doing NORMAL Read timeout processing, the device driver will wait as long as the value in the Read timeout word says to wait. The request will be completed after that interval of time elapses, if no more data has been received for the request. If any data is received by the device driver (from the receive hardware) for the request (including XON/XOFF characters), the specified period of time will be waited on again (for more data to arrive).

However, in the following two cases (of data being received), the current interval of time will continue to be waited on without starting to wait from the beginning of the interval again:

a. if input sensitivity using DSR is ENABLED and the value of the DSR modem control signal causes input data to be thrown away. Refer to Note 4: Input Sensitivity using DSR earlier in this chapter.
b. if null stripping is ENABLED and a null character is stripped. Refer to Note 6: Null Stripping earlier in this chapter.

If the device driver is doing NO WAIT Read timeout processing, the device driver will not wait for any data to be available in the receive queue. When the device driver begins to try to move data from the receive queue to the request, the request will complete. Whatever data is available in the receive queue at that time is the amount of data that will be moved to the request.

If the device driver is doing WAIT FOR SOMETHING Read timeout processing, the device driver will process the request initially as if it had NO WAIT timeout processing. If no data was available at the time the request would have completed due to NO WAIT processing, the request is not completed. Instead, the request waits for some data to be available before completing the request. However, the device driver does enter NORMAL Read timeout processing for this request. Therefore, if no data is available after the normal timeout processing interval then the request will be completed anyway. The request will never wait any longer then it would have due to normal Read timeout processing.

The Read timeout processing characteristics that apply to a given Read request is not determined until the device driver begins processing that request. Once the device driver begins processing that request, a change to the Read timeout processing characteristics of the device driver between WAIT FOR SOMETHING and NORMAL timeout processing may or may not take effect for the current Read request being processed. If the timeout period is changed by this IOCtl, the new timeout period may take effect immediately, or it may take effect after the next character is received from the receive hardware.

When the device driver initializes, NORMAL Read timeout processing is in effect.

When the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close - First Level Open), the value in the Read timeout word is set to 1 minute and NORMAL Read timeout processing characteristics is put into effect.

10. Extended hardware Buffering

    Extended Hardware Buffering refers to the capability of the COM port's serial controller device to buffer up to 16 characters in its internal hardware Receive and Transmit buffers. This buffering capability allows the device to relieve the operating system of the high overhead associated with servicing per-character receive and transmit hardware interrupts. On COM ports with a serial controller device that does not support Extended Hardware Buffering, bits 3 and 4 of DCB Flags3 will always be set to zero, indicating that the device is in Character Mode (Extended Hardware Buffering DISABLED).

    The ASYNC device driver defaults to automatically control the setting of Extended Hardware Buffering parameters using the Automatic Protocol Override mode of operation. This system default may be changed by manipulating bits in the Flags3 parameter of the Set Device Control Block Information IOCtl. The application or subsystem may establish manual control over this feature by setting Extended Hardware Buffering ENABLED, and manipulating the Receive Trigger Level and Transmit Buffer Load Count parameters. The application or subsystem may also set the serial device to run in Character Mode by setting Extended Hardware Buffering DISABLED. The three settings for this feature are described in detail as follows:

    **Automatic Protocol Override (System Default)**

    On any system configuration where one or more of the installed serial devices can correctly support Extended Hardware Buffering, the device driver will initialize that COM port to enable Automatic Protocol Override mode characteristics. This mode of the ASYNC device driver is defined with respect to the following device driver protocols:

    - Output Handshaking Using CTS, DSR, DCD
    - Automatic Transmit Flow Control
    - Input Sensitivity Using DSR.

    When any one or more of the above protocols is enabled, the Automatic Protocol Override feature will cause the Extended Hardware Buffering to not fully exploit the maximum potential performance benefit of the serial controller. Depending on which protocols are enabled, the device driver will automatically adjust either or both of the Receive Trigger Level and Transmit Buffer Load Count. The following descriptions identify the changes that each of the above protocols cause when Automatic Protocol Override mode is active.

    - Output Handshaking Using CTS, DSR (default ON).
    - Output Handshaking Using DCD (default OFF).

      When either of these handshaking protocols is enabled, the ASYNC device driver will (under Automatic Protocol Override) set the Transmit Buffer Load Count to 1. This means that it will service transmit interrupts one character at a time. The receive Trigger Level is not affected by these protocols in Automatic Protocol Override mode.

    - Automatic Transmit Flow Control (default OFF).

      When this protocol is enabled, the ASYNC device driver will (under Automatic Protocol Override) set the Receive Trigger Level to 1 and set the Transmit Buffer Load Count to 1. This means that it will service both Receive and Transmit interrupts on character at a time.

    - Input Sensitivity Using DSR (default ON).

      When this protocol is enabled, the Automatic Protocol Override feature of the ASYNC device driver will set the Receive Trigger Level to 1 by default on the respective COM port, forcing the device driver to service all characters received one character at a time. The Transmit Buffer Load Count is not affected by this protocol.

With all of the above listed protocols, Automatic Protocol Override allows the serial controller to remain in its FIFO mode setting, although it is not fully exploiting the potential performance benefit from the extended hardware buffering capability. The extended receive hardware buffering capability remains active so that receive hardware overrun errors are much less likely to occur.

**Note:** When Automatic Protocol Override mode is enabled, setting the DCB Flags3 bits for manipulating Receive Trigger Level and Transmit Buffer Load Count (bits 5,6,and 7) will have no effect. Automatic Protocol Override fully overrides any manual settings the application or subsystem may attempt to set.

**Note:** Having any of the above protocols enabled can significantly alter system performance characteristics with respect to activity on an asynchronous communications port. This is true particularly where the COM port is running at a high baud rate (2400 bps or higher), or where multiple COM ports are performing asynchronous I/O. This is a factor to consider when configuring a system to perform multiple serial port I/O, or when developing an application that requires high speed asynchronous communications. Disabling the above protocols, or setting the Device Control Block parameters to Extended Hardware Buffering ENABLED, will allow the device driver to fully exploit the serial controller's capability to its maximum benefit.

**Extended Hardware Buffering ENABLED**

Setting this mode will enable the user to fully utilize the Extended Hardware Buffering capabilities of the device driver. The Receive Trigger Level should be set to 8 (the device driver will receive 8 characters per interrupt) and the Transmit Buffer Load Count be set to 16 (the device driver will transmit 16 characters per interrupt).

Setting this mode in conjunction with any of the following device driver protocols can alter the behavior of that protocol in a significant way:

- Output Handshaking using CTS, DSR, DCD
- Automatic Transmit Flow Control
- Input Sensitivity using DSR.

For information on these protocols, refer to the description of these protocols in Chapter 6 (States of the ASYNC Device Driver).

**Extended Hardware Buffering DISABLED**

Setting this mode will completely disable the Extended Hardware Buffering capabilities of the serial port controller. This mode places the serial port device into character mode, meaning the device driver will service both transmit and receive interrupts one character at a time. This effectively eliminates any special considerations that might have been necessary when performing asynchronous communications with Extended Hardware Buffering ENABLED, or with Automatic Protocol Override mode active.

**Note:** On any COM port which is serviced by a serial controller that does not support FIFO mode, the device driver will always be set to Extended Hardware Buffering DISABLED. Attempts to set Extended Hardware Buffering ENABLED, or to enable Automatic Protocol Override mode, will be ignored (no error returned). Whenever the Return Device Control Block Information IOCtl is called, the bits in DCB Flags3 will identify the true state of Extended Hardware Buffering on these devices as Extended Hardware Buffering DISABLED.

**Note:** Setting the device driver to run with Extended Hardware Buffering DISABLED can significantly alter the system performance characteristics, particularly with respect to asynchronous communications I/O throughput. This is true especially where the COM port is running at a high baud rate (2400 bps or higher), or where multiple COM ports are performing asynchronous I/O.

11. Receive Trigger Level

The Receive Trigger Level represents the number of characters that must be received by a serial port controller that supports Extended Hardware Buffering before that device will generate a receive hardware interrupt. For example, with Extended Hardware Buffering ENABLED, assume the Receive Trigger Level is set to 8 characters. This means that a receive hardware interrupt will occur once for every 8 characters received at that COM port.

On COM ports with a serial controller device that does not support Extended Hardware Buffering, bits 5 and 6 of DCB Flags3 will always be set to zero, indicating that the device is in Character Mode (Receive Trigger Level = 1).

The default state for bits 5 and 6 of DCB Flags3 is zero, indicating that the Automatic Protocol Override mode is active and that other device driver protocols are enabled which force Automatic Protocol Override to keep Receive Trigger Level set to 1.

When the device driver is operating in Automatic Protocol Override mode, setting the Receive Trigger Level will have no effect. This parameter setting is completely ignored unless DCB Flags3 bits 3 and 4 are set to Extended Hardware Buffering ENABLED.

**Note:** System performance and asynchronous communications I/O throughput can be significantly degraded by setting Receive Trigger Level to 1 on COM ports whose serial controller device is capable of supporting Extended Hardware Buffering. This is particularly true where a COM port is sending and receiving data at high baud rates (over 2400 bps), or where multiple COM ports are simultaneously engaged in asynchronous communications.

**Note:** Setting the Receive Trigger Level to 14 characters can increase the probability of getting receive hardware overrun errors. This is particularly true where a COM port is sending and receiving data at high baud rates (over 2400 bps), or where multiple COM ports are simultaneously engaged in asynchronous communications.

The Return Device Control Block Information IOCtl will always give the actual current setting of the Receive Trigger Level, regardless of what the Extended Hardware Buffering mode is set to.

12. Transmit Buffer Load Count

    Transmit Buffer Load Count refers to the number of characters that the ASYNC device driver gives to a COM port's serial controller transmit hardware on the occurrence of a transmit hardware interrupt.

    On COM ports with a serial controller device that does not support Extended Hardware Buffering, bit 7 of DCB Flags3 will always be set to zero, indicating that the device is in Character mode (Transmit Buffer Load Count = 1).

    Normally, when the serial controller supports Extended Hardware Buffering, and the device driver protocols are set to fully utilize this hardware capability, the Transmit Buffer Load Count is set to 16 characters. This means that every time the device driver gets control of the serial controller on the occurrence of a transmit hardware interrupt, 16 characters are placed in the serial controller's transmit buffer.

    There may be situations where an application or communications subsystem needs to control the flow of data manually during a communications session. Setting the Transmit Buffer Load count to 1 character ensures that the device driver will get control of the serial controller every time a character is transmitted.

    When the device driver is operating in Automatic Protocol Override mode, setting the Transmit Buffer Load Count will have no effect. This parameter setting is completely ignored unless DCB Flags3 bits 3 and 4 are set to Extended Hardware Buffering ENABLED.

    **Note:** System performance and asynchronous communications I/O throughput can be significantly degraded by setting Transmit Buffer Load Count to 1 on COM ports whose serial controller device is capable of supporting Extended Hardware Buffering. This is particularly true where a COM port is sending and receiving data at high baud rates (over 2400 bps), or where multiple COM ports are simultaneously engaged in asynchronous communications.

The Return Device Control Block Information IOCtl will always give the actual current setting of the Transmit Buffer Load Count, regardless of what the setting of the Extended Hardware Buffering mode is set to.

# Category 1 –
# Function 61H

Return Baud (bit) Rate

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Bit Rate | WORD |

## Where

***Bit Rate***

A binary integer representing the actual baud (bit) rate of the COM device in bits per second, ROUNDED to the nearest whole number.

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned, the device driver will return the current baud (bit) rate of the COM device.

Return Line Characteristics (stop bits, parity, data bits, break)

## Parameter Packet Format

None. Packet Pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Data Bits | BYTE |
| Parity | BYTE |
| Stop Bits | BYTE |
| Transmitting Break | BYTE |

## Where

***Data Bits***

See set function 42H (Set Line Characteristics)

***Parity***

See set function 42H (Set Line Characteristics)

***Stop Bits***

See set function 42H (Set Line Characteristics)

***Transmitting Break***

0 means not currently transmitting break. 1 means currently transmitting break.

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

If a general failure is not returned, the device driver will return the line characteristics as defined.

# Category 1 – Function 64H

Return COM Status

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| COM Status Byte | BYTE |

## Where

***COM Status Byte***

1 The condition is true
0 The condition is false

| Note | Bit | Meaning |
|---|---|---|
| 3 | 0 | Tx waiting for CTS to be turned ON |
| 3 | 1 | Tx waiting for DSR to be turned ON |
| 3 | 2 | Tx waiting for DCD to be turned ON |
| 1 | 3 | Tx waiting because XOFF received |
| 2 | 4 | Tx waiting because XOFF transmitted |
| 5 | 5 | Tx waiting because break being transmitted |
| 6 | 6 | Character waiting to transmit immediately |
| 4 | 7 | Receive waiting for DSR to be turned ON |

Tx (transmit) status indicates why we may not be transmitting; regardless of whether there is data to transmit. However the device driver must be enabled for the given condition (for example, enabled for output handshaking for the modem control signal in question) for the status to reflect that the device driver would be waiting for the given condition to transmit.

For example, 00000001 means the device driver will put receive characters in the device driver receive queue, the device driver is not waiting to transmit a character immediately (Category 1 Function 44H), and we will not transmit characters from the device driver transmit queue because we are using CTS for output handshaking and CTS does not have the proper value.

Note:

1. See Set Device Control Block Note 2 (Category 1 Function 53H). and behave as if XOFF Received (Category 1 Function 47H).

   Characters will still be transmitted immediately (Category 1 Function 44H) and the device driver can still automatically transmit XONs and XOFFs because of automatic receive flow control (XON/XOFF) when the device driver is in this state.
2. See Set Device Control Block Note 2 (Category 1 Function 53H).

   Characters will still be transmitted immediately (Category 1 Function 44H) and the device driver can still automatically transmit XONs because of the automatic receive flow control when the device driver is in this state.
3. See Set Device Control Block Note 3 (Category 1 Function 53H).
4. See Set Device Control Block Note 4 (Category 1 Function 53H).
5. See Set break on (Category 1 Function 4BH).
6. See Transmit Byte Immediate (Category 1 Function 44H).

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported, and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned the device driver will return the COM device current status.

# Category 1 –
# Function 65H

Return Transmit Data Status

## Parameter Packet Format

None. Packet Pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Transmit Status | BYTE |

## Where

***Transmit Status***

Is returned as bit significant values. If the bit is 1, the condition is true. The bit is 0 if the condition is false. The number at the beginning of the description is the bit position number. The bit positions go from least to most significant.

0 - WRITE request packets in progress or queued.
1 - Data in the device driver transmit queue.
2 - The transmit hardware is currently transmitting data.
3 - Character waiting to be transmitted immediately.
4 - Waiting to automatically transmit an XON.
5 - Waiting to automatically transmit an XOFF.
6 - undefined
7 - undefined

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned the device driver will return the current transmit status of the COM device.

Return Modem Control Output Signals

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Modem Control Output Signals | BYTE |

## Where

***Modem Control Output Signals***

A bit value of 1 means the condition is ON. A bit value of 0 means the condition is OFF.

| *Bit* | *Meaning* |
|---|---|
| 0 | Data terminal ready (DTR) |
| 1 | Request to send (RTS) |
| 2 to 7 | Undefined |

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned, the device driver will return the current modem control output signals of the COM device.

# Category 1 –
# Function 67H

Return Current Modem Control Input Signals

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Modem Control Input Signals | BYTE |

## Where

***Modem Control Input Signals***

A bit value of 1 means the condition is ON. A bit value of 0 means that the condition is OFF.

| *Bit* | *Meaning* |
|---|---|
| 0 to 3 | Undefined |
| 4 | Clear To Send (CTS) |
| 5 | Data Set Ready (DSR) |
| 6 | Ring Indicator (RI) |
| 7 | Data Carrier Detect (DCD) |

## Returns

If the call is made with an invalid Parameter Packet value a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned, the device driver will return the current modem control input signals of the COM device.

Return Number of Characters in the Receive Queue

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Number of Characters Queued | WORD |
| Size of Receive Queue | WORD |

## Where

***Number of Characters Queued***

Binary integer with the number of received characters in the device driver receive queue. The device driver receive queue is a memory buffer in between the memory pointed to by the READ request packet and the receive hardware for this COM device. The application may not assume that there are no unsatisfied READ requests if there are characters in the device driver receive queue. The behavior of data movement between the READ request and the receive queue may change from release to release of the device driver. Applications should not be written to have a dependency on this information.

***Size of Receive Queue***

Binary integer with the size of the device driver receive queue. Applications should be written to be independent of the receive queue being of a fixed size. The information in this field allows the application to get the size of the receive queue. The current size of the receive queue is approximately 1K bytes but is subject to change.

Using this information, the application could be written to avoid device driver receive queue overruns. This could be done by using an application to application block protocol with the system that the application is communicating with.

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned, the device driver will return the information.

# Category 1 –
# Function 69H

Return Number of Characters in Transmit Queue

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Number of Characters Queued | WORD |
| Size of Transmit Queue | WORD |

## Where

***Number of Characters Queued***

Binary integer with the number of characters ready to be transmitted in the device driver transmit queue. The device driver transmit queue is a memory buffer between the memory pointed to by the WRITE request packet and the transmit hardware for this COM device. If the transmit queue is empty, the application may not assume that all WRITE requests have completed or that no WRITE requests are outstanding. The behavior of data movement between the WRITE request and the transmit queue may change from release to release of the device driver. Applications should not be written to be dependent on this information.

***Size of Transmit Queue***

Binary integer with the size of the device driver transmit queue. Applications should be written to be independent of the transmit queue being of a fixed size. The information in this field allows the application to get the size of the transmit queue. The size of the transmit queue is 128 bytes. Applications should not be written to have a dependency on this value as it is subject to change.

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

If a general failure error is not returned, the device driver will return the information.

Return COM Error (retrieve and then clear the COM device error information)

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| COM Error Word (COMERR) | WORD |

## Where

***COM Error***

The appropriate bits in the COM Error Word are set by the device driver when the events described below occur. The COM error word is not cleared unless this function is performed by the device driver or an OPEN request packet is received by the device driver and the COM device is not already open (from a previous open without a close first level open). See Note 5 of Set Device Control Block (Category 1 Function 53H).

| *Bit* | *Meaning* |
|---|---|
| 0 | Receive queue overrun. No room in the device driver receive queue to put a character read in from the receive hardware. |
| 1 | Receive hardware overrun. A character was not read from the hardware before the next character arrived causing a character to be lost. |
| 2 | The hardware detected a parity error. |
| 3 | The hardware detected a framing error. |
| 4 to 15 | Undefined |

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported, valid information is not returned in the Data Packet, and the COM error word is not cleared.

## Remarks

If a general failure error is not returned, the device driver will return and clear the COM device error information.

# Category 1 – Function 72H

Return COM Event Information (retrieve and then clear the COM device event word)

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| COM Event Word | WORD |

## Where

***COM Event Word***

The appropriate bits in the COM Event Word are set by the device driver when the following events occur:

**Note:** The COM Event Word is not cleared unless this function is performed by the device driver or an OPEN request packet is received by the device driver and the COM device is not already open (from a previous open without a close) (first level open).

| *Bit* | *Meaning* |
|---|---|
| 0 | Set when any character is read from the COM device receive hardware and placed in the receive queue. |
| 1 | Set whenever the serial port controller generates a Receive Timeout Interrupt during a receive request. This bit is always zero when the serial port controller does not support Extended Hardware Buffering. |
| 2 | Set when the last character in the device driver transmit queue is sent to the COM device transmit hardware. This does not mean that there is no data to send in any outstanding WRITE requests. |
| 3 | Set when the Clear to Send (CTS) signal changes state. |
| 4 | Set when the Data Set Ready (DSR) signal changes state. |
| 5 | Set when the Data Carrier Detect (DCD) signal changes state. |
| 6 | Set when a break is detected. |
| 7 | Set when a parity, framing or overrun error occurs (an overrun can be a receive hardware overrun or a receive queue overrun). |
| 8 | Set when trailing edge of Ring Indicator is detected. |
| 9 to 15 | Undefined |

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported, valid information is not returned in the Data Packet, and the event word is not cleared.

## Remarks

If a general failure is not returned, the device driver will return the current value of the event word and then clear it.

Return Device Control Block (DCB) Information

## Parameter Packet Format

None. Packet pointer must be NULL.

## Data Packet Format

| Field | Length |
|---|---|
| Write Timeout | WORD |
| Read Timeout | WORD |
| Flags1 | BYTE |
| Flags2 | BYTE |
| Flags3 | BYTE |
| Error Replacement Character | BYTE |
| Break Replacement Character | BYTE |
| XON Character | BYTE |
| XOFF Character | BYTE |

## Where

***Write Timeout***

specifies the time period used for write timeout processing. Refer to Category 1 Function 53H Note 8: Write Timeout earlier in this chapter. The value is in .01 second units based on zero, when zero equals .01 seconds.

***Read Timeout***

specifies the time period used for read timeout processing. Refer to Category 1 Function 53H Note 9: Read Timeout earlier in this chapter. The value is in .01 second units based on zero, when zero equals .01 seconds.

***Flags1***

| *Bit* | *Meaning* |
|---|---|
| 0-1 | DTR Control Mode<br>*Bit 1* *Bit 0*<br>0 0 Disable<br>0 1 Enable<br>1 0 Input handshaking |
| 2 | Reserved (returned as zero) |
| 3 | Enable output handshaking using CTS |
| 4 | Enable output handshaking using DSR |
| 5 | Enable output handshaking using DCD |
| 6 | Enable input sensitivity using DSR |
| 7 | Reserved (returned as zero) |

***Flags2***

| *Bit* | *Meaning* |
|---|---|
| 0 | Enable automatic transmit flow control (XON/XOFF) |
| 1 | Enable automatic receive flow control (XON/XOFF) |
| 2 | Enable error replacement character |
| 3 | Enable null stripping (remove null bytes) |
| 4 | Enable break replacement character |

| | | | |
|---|---|---|---|
| 5 | Automatic receive flow control<br>0 = Normal<br>1 = Full-Duplex | | |
| 6-7 | RTS Control Mode | | |
| | ***Bit 7*** | ***Bit 6*** | |
| | 0 | 0 | Disable |
| | 0 | 1 | Enable |
| | 1 | 0 | Input handshaking |
| | 1 | 1 | Toggling on transmit |

***Flags3***

| ***Bit*** | ***Meaning*** | | |
|---|---|---|---|
| 0 | Enable write infinite timeout processing | | |
| 1-2 | Read timeout processing | | |
| | ***Bit 2*** | ***Bit 1*** | |
| | 0 | 1 | Normal read timeout processing |
| | 1 | 0 | Wait for something, read timeout processing |
| | 1 | 1 | No wait, read timeout processing |
| 3-4 | Extended Hardware Buffering | | |
| | ***Bit 4*** | ***Bit 3*** | |
| | 0 | 0 | Extended Hardware Buffering DISABLED |
| | 0 | 1 | Extended Hardware Buffering ENABLED |
| | 1 | 0 | Automatic Protocol Override |
| 5-6 | Receive Trigger Level | | |
| | ***Bit 6*** | ***Bit 5*** | |
| | 0 | 0 | 1 character |
| | 0 | 1 | 4 characters |
| | 1 | 0 | 8 characters |
| | 1 | 1 | 14 characters |
| 7 | Transmit Buffer Load Count<br>0 = 1 character<br>1 = 16 characters | | |

See function 53H, Set Device Control Block (DCB) for field definitions.

***Error Replacement Character***
Returned value. Refer to Category 1 Function 53H Note 5: Error Replacement Character earlier in this chapter.

***Break Replacement Character***
Returned value. Refer to Category 1 Function 53H Note 7: Break Replacement Character earlier in this chapter.

***XON Character***
Returned value. Refer to Category 1 Function 53H Note 2: Automatic Flow Control earlier in this chapter.

***XOFF Character***
Returned value. Refer to Category 1 Function 53H Note 2: Automatic Flow Control earlier in this chapter.

## Returns

If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

## Remarks

The general Device Control Block (DCB) parameter access functions (53H and 73H) are used for:

- Automatic transmit flow control (start/stop transmit when XON/XOFF character received)
- Automatic receive flow control (transmit XON/XOFF when receive buffer fills/empties)
- Determine XON/XOFF characters
- DTR control mode (enable/disable/input handshaking)
- RTS control mode (enable/disable/input handshaking/toggling on transmit)
- Output handshaking using CTS/DSR/DCD (control signal determines when to transmit)
- Input sensitivity using DSR (reception of data controlled by DSR)
- Error replacement character and processing
- Break replacement character and processing
- Null stripping
- Receive/transmit timeout processing
- Extended Hardware Buffering control.

If a general failure error is not returned, the device driver will return valid information in the Data Packet.

**Note:** To maintain upward compatibility, it is the responsibility of the application to do a Return Device Control Block (DCB) information BEFORE doing a Set Device Control Block (DCB). The appropriate information in the returned control block should be modified by the application and then the Set function can be done.

The bit fields that are labeled Reserved will be returned as 0, but applications should not be written to make this assumption. Applications also should not be written to assume that the fourth bit combination will never be returned for DTR Control Mode, Read Timeout processing, or Extended Hardware Buffering. Applications should not attempt to manipulate these Reserved bits.

# Category 3 Pointer Draw Control IOCtl Commands

Following is a summary of Category 3 descriptions:

| *Function* | *Description* |
|---|---|
| 72H | Get pointer draw address |

Get pointer draw address

## Parameter Packet Format

None

## Data Packet Format

| Field | Length |
|---|---|
| Return Code | WORD |
| Pointer Draw Routine Entry Point (Selector:Offset) | DWORD |
| Pointer Draw Routine Data Segment Selector | WORD |

## Returns

Information in data packet above.

## Remark

The call is used by the mouse subsystem to obtain the entry point address of the pointer draw routine. The pointer draw routine is contained within the pointer draw device driver. It is called by the mouse device driver to update the pointer image on the screen. The far address returned by function code 72H is passed by the mouse subsystem to the mouse device driver through an IOCtl interface (reference the mouse control IOCtl commands, category 07H). The mouse device driver saves the far address passed and uses it when calling the pointer draw routine.

This function is supported by the pointer draw device driver.

# Category 3 Video Control IOCtl Commands

The following is a summary of Category 3, Video Control IOCtl descriptions:

| *Function* | *Description* |
|---|---|
| 73H | Initialize Call Vector Table |

Initialize call vector table

## Parameter Packet Format

None

## Data Packet Format

| Field | Length |
|---|---|
| Far Address of the Call Vector Table | DWORD |

## Returns

None

## Remark

See the description of the Call Vector Table in the description of video device handler initialization.

# Category 3 Screen Control IOCtl Commands

The following is a summary of Category 3, Screen Control IOCtl descriptions:

| *Function* | *Description* |
|---|---|
| 70H | Allocate an LDT selector |
| 71H | Deallocate an LDT selector |
| 74H | ABIOS pass-through |
| 75H | Allocate an LDT selector with Offset |

Allocate an LDT selector

## Parameter Packet Format

| Field | Length |
|---|---|
| 32-bit physical address | DWORD |
| Length | WORD |

## Data Packet Format

| Field | Length |
|---|---|
| LDT Selector | WORD |

## Returns

None

## Remark

This function is supported by the screen device driver.

A 32-bit physical address and 16-bit length are passed to the device driver. The device driver returns an LDT selector for that area of memory. Read/Write access is granted to any data areas completely contained in the range of addresses from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses from 00000 to FFFFF. Requests for any other data areas will result in a return code of ERROR_I24_INVALID_PARAMETER.

If the PhysToUVirt DevHelp function returns a non-zero offset along with the requested selector, the selector will be deallocated and ERROR_I24_INVALID_PARAMETER will be returned. In this case, the Allocate LDT Selector with Offset IOCtl (75H) should be used.

# Category 3 – Function 71H

Deallocate an LDT selector.

## Parameter Packet Format

| Field | Length |
|---|---|
| LDT Selector | WORD |

## Data Packet Format

None

## Returns

None

## Remark

This function is supported by the screen device driver.

An LDT selector previously created by the screen device driver, by way of the Category 3, Function 70H IOCTL to Allocate an LDT selector, is passed in the parameter block. The selector is deallocated.

Pass an ABIOS request block to unit 0 of the video device opened by SCREEN$ and return it to the caller on completion of the request.

## Parameter Packet Format

| Field | Length |
|---|---|
| ABIOS Request Block | OTHER |

## Data Packet Format

None.

## Returns

None

## Remark

This function is supported by the screen device driver.

The parameter packet is copied to a 64-byte work area that is used as an ABIOS request block. The LID and UNIT fields are overwritten with the logical ID of the ABIOS video device opened by SCREEN$ and 0, respectively. This request block is used with DevHlp_ABIOSCall and the resulting request block is copied back to the caller's parameter packet.

# Category 3 –
# Function 75H

Allocate an LDT selector with Offset

## Parameter Packet Format

| Field | Length |
|---|---|
| 32-bit physical address | DWORD |
| Length | WORD |

## Data Packet Format

| Field | Length |
|---|---|
| Offset within New LDT Selector | WORD |
| New LDT Selector | WORD |

## Returns

None

## Remark

This function is supported by the screen device driver.

A 32-bit physical address and 16-bit length are passed to the device driver. The device driver returns an LDT selector and offset for that area of memory. Read/Write access is granted to any data areas completely contained in the range of addresses from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses from 00000 to FFFFF. Requests for any other data areas will result in a return code of ERROR_I24_INVALID_PARAMETER.

# Category 4 Keyboard Control IOCtl Commands

The following is a summary of Category 4 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 50H | Set code page |
| 51H | Set input mode (default ASCII) |
| 52H | Set interim character flags |
| 53H | Set shift state |
| 54H | Set typamatic rate and delay |
| 55H | Notify of change of foreground session |
| 56H | Set session manager Hot Key |
| 57H | Set KCB |
| 58H | Set code page ID |
| 59H | Set Read Notification |
| 5AH | Alter Keyboard LEDs |
| 5BH | Reserved |
| 5CH | Set NLS & custom code page |
| 5DH | Create a new logical keyboard |
| 5EH | Destroy a logical keyboard |
| 71H | Get input mode |
| 72H | Get interim character flags |
| 73H | Get shift state |
| 74H | Read character data record(s) |
| 75H | Peek character data record |
| 76H | Get session manager Hot Key |
| 77H | Get keyboard type |
| 78H | Get code page ID |
| 79H | Translate scan code to ASCII |
| 7AH | Get Keyboard H/W ID |
| 7BH | Get Keyboard Code Page Support Info |

# Category 4 – Function 50H

Set Code Page

## Parameter Packet Format

| Field | Length |
|---|---|
| Code Page Translation Table Pointer | DWORD |

## Data Packet Format

None

## Where

***Code Page Translation Table Format***

has the following format where there are 127 copies of the KeyDef rec below (includes 1 for each possible scan code that may be returned from the keyboard). Not all entries are used; unused entries are zero. The entries are in scan code order, based on the remapped scan codes returned by the keyboard controller when it is in the DOS mode. The DOS mode translates keyboard scan codes to scan codes based on the position of the keys as they are on the standard PC keyboard (plus additional keys on the Enhanced Keyboard). The DOS mode also converts key "break" codes to the equivalent scan code with the high order bit turned on (that is, adds 128 to the code).

```
XlateTable:
  XHeader    : XHeader
  KeyDef1    : KeyDef
  KeyDef2    : KeyDef
  KeyDef3    : KeyDef
     .          .
     .          .
     .          .
  KeyDef127  : KeyDef
  AccentTbl  : AccentTable
End XlateTable

XHeader:
  XTableID          : Word       [Code Page ID               ]
  XTableFlags1      : Rec[Word Width]
                    : The following three bits determine which
                    : shift key or key combination affects CHAR3
                    : of each KeyDef.
     ShiftAlt       : Bit 0 [Use Shift-Alt instead of Ctrl-Alt]
     AltGrafL       : Bit 1 [Use left Alt key as Alt-Graphics]
     AltGrafR       : Bit 2 [Use right Alt key as Alt-Graphics]
     ShiftLock      : Bit 3 [Treat Caps Lock As Shift Lock    ]
     DefaultTable   : Bit 4 [Default table for the Lang.]
     ShiftToggle    : Bit 5 [Toggle or Latch ShiftLock]
                    :       When 1 toggle else latch
     AccentPass     : Bit 6 [Pass accent and non-accent key through]
                    :       When 1 pass on accent keys and beep,
                    :       else beep only.
                    : The following four bits determine which
                    : shift key or key combination causes Char5
                    : to be used in each KeyDef.
     CapsShift      : Bit 7 [Caps-Shift uses CHAR5]
     MachDep        : Bit 8 [Machine dependent table]
     Reserved       : Bit 9-10
     Reserved       : Bit 11 - 15
  EndRec XtableFlags1
  XTableFlags2      : Rec[Word Width]
     Reserved       : Bit 0 - 15
  EndRec XtableFlags2
  KbdType           : Word      [Keyboard type, see below  ]
  KbdSubType        : Word      [Reserved.                 ]
  XtableLen         : Word      [Length of Table           ]
  EntryCount        : Word      [Number of KeyDef entries  ]
```

```
  EntryWidth        : Word      [Width of KeyDef entries   ]
  Country           : Word      [Language ID               ]
  TableTypeID       : Word      [Identifies the table type ]
                          1st byte (type):   01X  00X
                        2nd byte (sub-type):   00X  Reserved
  SubCountryID      : 4 Bytes   [Sub-Language Identifier   ]
  Reserved          : 8 Words   [Reserved.                 ]
End XHeader
KeyDef = Rec                    [127 copies of this record.]
  XlateOp = Rec [word field]    [Translate operation specifier.]
    AccentFlags : 7 Bits        [See notes 1 and 8, below.]
    KeyType     : 9 bits        [Note 2, below.]
  Char1 : Byte                  [Use depends on KeyType, below.]
  Char2 : Byte                  [Use depends on KeyType, below.]
  Char3 : Byte                  [Use depends on KeyType, below.]
  Char4 : Byte                  [Use depends on KeyType, below.]
  Char5 : Byte                  [Use depends on KeyType, below.]
EndRec KeyDef
AccentTable = Rec               [Table of accent key definitions.]
  AccentEntry1 : AccentEntry
  AccentEntry2 : AccentEntry
     .          .
     .          .
     .          .
  AccentEntry7 : AccentEntry
EndRec AccentTable


AccentEntry = Rec     [Accent entry definition. See notes 1 and 9.]
  NonAccent : 2 Bytes [Char/scan code when not used as accent]
  CtlAccent : 2 Bytes [Char/scan code when used with CTL.     ]
  AltAccent : 2 Bytes [Char/scan code when used with Alt.     ]
  Map1  : 2 Bytes   [From char to char for translation. ]
  Map2  : 2 Bytes     "    "        "            "
     .          .    "    "        "            "
     .          .    "    "        "            "
     .          .    "    "        "            "
  Map20 : 2 Bytes     "    "        "            "
EndRec AccentEntry

TableTypeID
          1st byte     2nd byte
          type         sub-type
          00X          Reserved
OS/2      01X          00X
```

## Notes about the Code Page

1. The AccentFlags field of the KeyDef record has seven flags that are individually set if a corresponding entry in the accent table applies to this scan code. If the key pressed immediately before the current one was an accent key, and the bit for that accent is set in the AccentFlags field for the current key, the corresponding AccentTable entry is searched for the replacement character value to use. If no replacement is found and Bit 6 of the XlateFlags1 field is set, the "not-an-accent" beep is sounded and the accent character and current character are passed as two separate characters. Also see note 8.
2. The KeyType field of the KeyDef record currently has the following values defined. The remaining values up to 1FH are undefined. In the following table the effect of each type of shift is defined. Except where otherwise noted, when no shifts are active, Char1 is the translated character. References to undefined are clarified in note 3 below. Note that either the Alt, AltC/S/G or both may be present on a keyboard based on the AltGrafL and AltGrafR bits in the XTableFlags1 flagword in the table header.
   - 01H) AlphaKey - Alphabetical character key:

     ```
     SHIFT   - Uses Char2 (If CAPSLOCK, uses Char1).
     CAPSLOCK- Uses Char2 (If SHIFT, uses Char1).
     CTL     - Set standard "control" code for this key's
               Char1 value (see note 4 below).
     ALT     - Standard "extended" code (see note 7).

     ALTC/S/G - Uses Char 3 if it is not zero
     ```

## Category 4 – Function 50H

- 02H) SpecKey - Special non-alphabetical character key, no CapsLock or Alt:

```
SHIFT   - Uses Char2.
CAPSLOCK- No effect, only depends on SHIFT or CTL.
CTL     - See note 4 below.
ALT     - Marked undefined.

ALTC/S/G - Uses Char 3 if it is not zero
```

- 03H) SpecKeyC - Special non-alpha character key with CapsLock See note 15.

```
SHIFT   - Uses Char2 (If CAPSLOCK, uses Char1).
CAPSLOCK- Uses Char2 (If SHIFT, uses Char1).
CTL     - See note 4 below.
ALT     - Use Char4 if not zero.  See also note 7.

ALTC/S/G - Uses Char 3 if it is not zero
```

- 04H) SpecKeyA - Special non-alpha character key, with Alt (no CapsLock):

```
SHIFT   - Uses Char2
CAPSLOCK- No effect, only depends on SHIFT, CTL or ALT.
CTL     - See note 5 and note 9 below.
ALT     - See note 7 and note 10 below.

ALTC/S/G - Uses Char 3 if it is not zero
```

- 05H SpecKeyCA - Special non-alpha character key, with CapsLock & Alt:

```
SHIFT   - Uses Char2 (If CAPSLOCK, uses Char1).
CAPSLOCK- Uses Char2 (If SHIFT, uses Char1).
CTL     - See note 4 below.
ALT     - See note 7 below.

ALTC/S/G - Uses Char 3 if it is not zero
```

- 06H) FuncKey - Function keys (Char1 = "n" in "Fn", Char2 ignored, sets "extended" codes 58+Char1 if no shift or if F11 or F12, uses 139 and 140).

```
SHIFT   - Sets "extended" codes 83+Char1;
          F11 and F12 use 141 and 142 respectively
CAPSLOCK- No effect on function keys.
CTL     - Sets "extended" codes 93+Char1;
          F11 and F12 use 143 and 144 respectively
ALT     - Sets "extended" codes 103+Char1;
          F11 and F12 use 145 and 146 respectively.

ALTC/S/G - Uses Char 3 if it is not zero
```

- 07H) PadKey - Keypad keys (see note 5 for definition of Char1, and note that non-shifted use of these keys is fixed to the "extended" codes):

```
SHIFT   - Uses Char2 (Unless NUMLOCK, then see note 5).
CAPSLOCK- No effect on pad keys (NUMLOCK does, note 5).
CTL     - Sets "extended" codes (see note 5).
ALT     - Used to "build" a character (see note 5).

ALTC/S/G - Uses Char 3 if it is not zero
```

- 08H) SpecCtlKey - "Action" keys that do special things with Ctrl down:

```
SHIFT   - No effect on these keys.
CAPSLOCK- No effect on these keys.
CTL     - Uses Char2.
ALT     - See note 7.

ALTC/S/G - Uses Char 3 if it is not zero
```

- 09H) PrtSc - Print Screen key (sets Char1 normally; see also note 17):

```
SHIFT    - Signal the Print Screen function.
CAPSLOCK- No effect on this key.
CTL      - Sets extended code and signals
           the Print Echo function.
ALT      - Marked undefined.

ALTC/S/G - Uses Char 3 if it is not zero
```

- 0AH) SysReq - System Request key; treated like a shift key. (See note 6 below).
- 0BH) AccentKey - Keys that affect the "next" key pressed (also known as dead keys). Char1 is an index into the AccentTbl field of the XlateTable, selecting the AccentEntry that corresponds to this key. Char2 and Char3 do the same for the shifted Accent character. See note 15.

```
SHIFT    - Use Char2 to index to applicable AccentEntry.
CAPSLOCK- No effect on this key.
CTL      - Use CtlAccent character from AccentEntry
           (see note 8).
ALT      - Use AltAccent character from AccentEntry
           (see note8).

ALTC/S/G - Use Char3 to index to applicable AccentEntry.
```

**Note:** Key types 0CH through 13H set Char1 & Char2 to mask values as defined in note 6 below.

- 0CH) ShiftKeys - SHIFT or Ctrl key, sets/clears flags. Char1 holds the bits in the lower byte of the shift status word to set when the key is down and clear when the key is released. Char2 does the same thing for the upper byte of the shift status word, unless the "secondary" key prefix (hex E0) is seen immediately prior to this key, in which case Char3 is used in place of Char2.
- 0DH) ToggleKey - General toggle key (like Caps Lock). Char1 holds the bits in the lower byte of the shift status word to toggle on the first make of the key after it is pressed. Char2 holds the bits in the upper byte of the shift status word to set when the key is down and clear when the key is released unless the "secondary" key prefix (hex E0) is seen immediately prior to this key, in which case Char3 is used in place of Char2.
- 0EH) AltKey - Alt key. Treated just like ShiftKeys above, but has its own key type because when seen, the accumulator used for Alt-Padkey entry is zeroed to prepare such entry. See note 5 for more information about Alt-PadKey entry. Sometimes this key is treated as "AltC/S/G" key if one of the AltGraf bits is on in XTableFlags1.
- 0FH) NumLock - NUMLOCK key. Behaves like ToggleKey normally, but the base keyboard device driver will set a pause screen indication when this key is seen along with the Ctrl key depressed. The pause is cleared on the following keystroke, if that stroke is a character generating key.
- 10H) Caps Lock - The Caps Lock key. This key is treated exactly like a type 0DH toggle key. It has a separate entry here so that if it can be processed like a shift lock key when that flag is set in the XTableFlags1 word in the header. When treated as a ShiftLock, the Caps Lock flag in the shift status word is set ON on any make of this key, and only cleared when the left or right shift key is depressed. Char2 and Char3 are processed the same as ToggleKey.
- 11H) ScrollLock - SCROLL LOCK key. Behaves like ToggleKey normally, but has a separate entry here so that when used with "Ctrl-" it can be recognized as "Ctrl-Break".
- 12H) XShiftKey - Extended Shift Key (for Country Support). See note 9 for more information.
- 13H) XToggleKey - Extended Toggle Key (for Country Support). See note 9 for more information.
- 14H) SpecKeyCS - Special key 1 for foreign keyboard processing. See note 15 for more information.

```
SHIFT     - Use Char2
CAPSLOCK  - Use Char4
CTL       - See note 4 below.
ALT       - See note 7.

ALTC/S/G  - Use Char3

CAPS+SHFT - Use Char5
```

- 15H) SpecKeyAS - Special key 2 for foreign keyboard processing. See note 15 for more information.

```
SHIFT       - Use Char2
CAPSLOCK    - No effect on this key
CTL         - See note 4 below.
ALT         - Use Char4. See note 14 below

ALTC/S/G    - Use Char3. See note 14 below
```

- 16-1FFH) Reserved

3. Undefined Character Code: Any key combination that doesn't fall into any of the defined categories (for example, the Ctrl key pressed along with a key that has no defined control mapping) will be mapped to the value 0 and the keytype will be set in the KeyPacket record indicating undefined translation. The KeyPacket record passed to the monitors (if any are installed) will contain the original scan code in the ScanCode field and the 0 in the Character field for this key. Note that no character data records with an undefined character code will be placed in the keyboard input buffer.
4. Ctrl Key Notes: There are six possible situations for when a key is pressed along with only the Ctrl shift key. They are:
   a. The key pressed is an AlphaKey character. In this case the Ctrl plus Char1 combination defines one of the standard defined control codes. They are (all numbers are decimal):

| Ctrl- | Mapping | Code Name | Ctrl- | Mapping | Code Name |
|---|---|---|---|---|---|
| a | 1 | SOH | n | 14 | SO |
| b | 2 | STX | o | 15 | SI |
| c | 3 | ETX | p | 16 | DLE |
| d | 4 | EOT | q | 17 | DC1 |
| e | 5 | ENQ | r | 18 | DC2 |
| f | 6 | ACK | s | 19 | DC3 |
| g | 7 | BEL | t | 20 | DC4 |
| h | 8 | BS | u | 21 | NAK |
| i | 9 | HT | v | 22 | SYN |
| j | 10 | LF | w | 23 | ETB |
| k | 11 | VT | x | 24 | CAN |
| l | 12 | FF | y | 25 | EM |
| m | 13 | CR | z | 26 | SUB |

   Note that any key defined as AlphaKey will use the Char1 code value minus 96 (ASCII code for "a") plus 1 to set the mapping shown above. So any scan code defined as AlphaKey must assign to Char1 one of the allowed lower case letters.
   b. The key pressed is a non-alpha character (like "["), but is not an "action" key (like Enter, Backspace, or an arrow key). This is a SpecKey[C][A] in the list of key types above. In this case (with one exception) the mapping is based on the scan code of the key. Though the key may be re-labeled, the Ctrl+Char combination is always mapped based on the scan code of the key using the following table (all numbers are decimal):

| Scan Code | US Kbd Legend | Mapped Value | Name of New Code | |
|---|---|---|---|---|
| 3 | 2 @ | 0 | Null | |
| 7 | 6 ^ | 30 | RS | |
| * 12 | - _ | 31 | US | (see note below) |
| 26 | [ { | 27 | Esc | |
| 27 | ] } | 29 | GS | |
| 43 | \ \| | 28 | FS | |

   **Note:** The mapping for the hyphen character ("-") is the one exception. The scan code for it is ignored, only the ASCII code for hyphen (decimal 45) is looked for (in Char1) when

mapping the Ctrl + "-" combination. This is because there may be more than one occurrence of the "-" key on the keyboard. The Ctrl + Keypad "-" combination produces character/scan code values of 00/8EH, respectively.

c. The key pressed is an "action" key like Enter, backspace, or an arrow key. These keys generate special values when used in conjunction with the Ctrl key. Those actions are defined in other notes where they apply. Two particular keys in this category are:
   Ctrl + Enter = LF(010)
   Ctrl + Backspace = Del(127)
d. The key pressed is a function key (F1 - F12). See the FuncKey description in note 2.
e. The key pressed is an accent key. See note 8 for details.
f. The key is not defined in conjunction with Ctrl. In that case, the key will treated as undefined, as described in note 3.

5. Pad Key Notes: The pad keys have several uses depending on various shift states. Some of them are based on their position on the keyboard. Because keyboard layouts change, here are the hard coded assumed positions of the keypad keys, with the "offset" value that must be coded into Char1 of the table. Any remapping must use the Char1 values defined below for the keys that correspond to the pad keys given by the Legend or Char2 values shown:

| US Kbd Legend | | Scan Code | Char1 REQUIRED | Char2 US kbd | With Ctrl |
|---|---|---|---|---|---|
| Home | 7 | 71 | Decimal 0 | ASCII 7 | Decimal 119 |
| Up | 8 | 72 | " 1 | " 8 | " 141 |
| PgUp | 9 | 73 | " 2 | " 9 | " 132 |
| | - | 74 | " 3 | " - | " 142 |
| Left | 4 | 75 | " 4 | " 4 | " 115 |
| | 5 | 76 | " 5 | " 5 | " 143 |
| Right | 6 | 77 | " 6 | " 6 | " 116 |
| | + | 78 | " 7 | " + | " 144 |
| End | 1 | 79 | " 8 | " 1 | " 117 |
| Down | 2 | 80 | " 9 | " 2 | " 145 |
| PgDn | 3 | 81 | " 10 | " 3 | " 118 |
| Ins | 0 | 82 | " 11 | " 0 | " 146 |
| Del | . | 83 | " 12 | " . | " 147 |

Note that when NumLock is OFF, or if Shift is active & NumLock ON, the code returned is the "extended" code. The code returned corresponds to the Legends above (Home, PgUp, etc). When NumLock is ON, or if SHIFT is active & NumLock is OFF, the code returned is Char2. Note that the " + " and "-" keys will also return Char2 when the shift key is down.

When the Alt key is used with the PadKeys, the absolute value of the pressed key (looked up using the required Char1 value) is added to the accumulated value of any of the previous numeric keys pressed without releasing the Alt key. Before adding the new number to the accumulated value, that accumulation is multiplied by ten, with overflow beyond 255 ignored. When Alt is released, the accumulation becomes a Character code, and is passed along with a scan code of zero. Note that if any key other than the 10 numeric keys is hit, the accumulated value is reset to zero. When the keypad "*", "-", or " + " keys are pressed while the Alt key is down, the extended characters 55, 74, and 78 (decimal) are returned, respectively.

When AltGraphics is used with the PadKeys the Char3 value is returned if it is non-zero and if an AltGraf bit is set in XTableFlags1; otherwise it is treated the same as the Alt key.

On the enhanced keyboard, the secondary keypad keys return as an extended character the scan code of the key plus 80 (decimal) when pressed in conjunction with the Alt key. The secondary "/" key returns an extended character of 164 when pressed in conjunction with the Alt key.

6. State Key Notes: Each state key entry has Char1, Char2, and Char3 defined as follows:

Char1 is a mask to set the appropriate bit in the low byte of the keyboard Shift Flags when the state key is pressed. When the state key is a toggle key, the set bit will be toggled each additional time the key is pressed. When the state key is not a toggle key, the set bit will be cleared when the key is released.

Char2 is a mask to set the appropriate bit in the high byte of the Keyboard Shift Flags when the key is pressed.

Char3 is used in place of Char2 when the secondary key prefix is seen immediately prior to this key.

The masks are (numbers are in hex):

| Key | Char1 | Char2 | Char3 |
|---|---|---|---|
| Right Shift | 01 | 00 | 00 |
| Left Shift | 02 | 00 | 00 |
| Ctrl Shift | 04 | 01 | 04 |
| Alt Shift | 08 | 02 | 08 |
| Scroll Lock | 10 | 10 | 10 |
| Num Lock | 20 | 20 | 20 |
| Caps Lock | 40 | 40 | 40 |
| SysReq | 00 | 80 | 80 |

Note that the INS key is not treated as a state key, but as a Pad key. Also note that SysReq is included here as it is treated as a shift key.

7. Alt Character Notes: Most of the keys defined in a category that allows the Alt key (AlphaKey, SpecKeyA, SpecKeyCA) return a value called an Extended Character. This value is a character code of 00H or E0H with a second byte (using the ScanCode field of the CharData record) defining the extended code. In most cases this value is the scan code of the key. Since the legend on these keys may be remapped on a foreign language keyboard, the Alt based extended code is hard to define in a general sense. The following rules are used:
   - AlphaKey: The extended code is derived from Char1 (the "lower case" character) as it was originally mapped on the PC keyboard. The original scan code value itself is the extended code that a character will return. These keys can be moved and will still return their original Alt extended codes.
   - SpecKeyA and SpecKeyCA: This category is used for all keys that are not an alphabetical character or an "action" code (like Enter or Backspace, the only exception being the tab key which IS treated as a character). On foreign keyboards these keys may be moved around and/or have new values assigned to them (such as special punctuation symbols). So the Alt mappings must be based on the real scan code. The effect of this is that keys defined by the SpecKey__ classification will only have an Alt mapping if it is in one of the positions defined below. In that case the Alt extended code is as shown in the table.

| Scan Code | US Kbd Legend | Alt Value | Scan Code | US Kbd Legend | Alt Value | |
|---|---|---|---|---|---|---|
| 2 | 1 ! | 120 | 15 | Tab | 165 | |
| 3 | 2 @ | 121 | 26 | [ { | 26 | |
| 4 | 3 # | 122 | 27 | ] } | 27 | |
| 5 | 4 $ | 123 | 28 | Enter | 28 | |
| 6 | 5 % | 124 | 39 | ; : | 39 | |
| 7 | 6 ^ | 125 | 40 | ' " | 40 | |
| 8 | 7 & | 126 | 41 | ' ~ | 41 | |
| 9 | 8 * | 127 | 43 | \ \| | 43 | (equals W.T.C. key number 42) |
| 10 | 9 ( | 128 | 51 | , < | 51 | |
| 11 | 0 ) | 129 | 52 | . > | 52 | |
| 12 | - _ | 130 | 53 | / ? | 53 | |
| 13 | = + | 131 | | | | |

   The secondary "/" key returns an extended character of 164 when pressed while Alt is down.
   - FuncKey: Defined in note 2.
   - SpecCtlKey: The Alt+ values of the Escape, Backspace, and Enter keys are extended characters equalling 1, 14, and 28 (decimal), respectively.

   When AltGraphics is used the Char3 value is returned if it is non-zero and if an AltGraf bit is set in XTableFlags1; otherwise it is treated the same as the Alt key.
8. Accent Key Notes: When an accent key is pressed along with Ctrl or Alt it is treated as a regular key. The character it is translated to is the one in the CtlAccent or AltAccent field of the AccentEntry pointed to by the Char5 value of the KeyDef. If the key being defined will have no defined value with Ctrl or Alt, it should have zeroes in the field of the undefined combination.

   When an accent key is pressed by itself (or with Right or Left Shift or AltGraphics) it will not be translated immediately. The Char1 (or Char2 when Left or Right Shift is used or when the AltGraphics is used) index in the KeyDef record will be used with the next key received to check

if the next key has an accent mapping. If that next key has no mapping for this accent (that is, if it has no bit set in its AccentFlags for this accent, or if that next key is not found in this accent's AccentEntry) then the character value in the NonAccent field of the AccentEntry is used as the character to display, followed by the translation of that next key (that is, both characters are passed on) after the not-an-accent beep is sounded.

Note that if a key doesn't change when a left or right shift key is held down it should use the same value for Char1 and Char2 so that the accent will apply in both the shifted and non-shifted cases. If the accent value is undefined when used with a shift key or AltGraphics the value in Char2 or Char3 should be zero.

Any accent key that doesn't have an Alt or Ctrl mapping should put zeros in the AltAccent and CtlAccent fields of its AccentEntry. If the value in the table is between 1 and 7 then the key is considered an accent key and further accent key processing is indicated. Reference note 1 for further information.

9. Extended State Key Notes: For special Country support, the Keyboard Device Driver maintains another byte of shift status. Key types 12H and 13H are provided for manipulation of that byte. The other fields of the KeyDef are:
   - Char1: A mask where the bits that are on are those bits that define the field being used for the Char2 value. Only the bits in the NLS shift status byte that correspond to the bits in this byte will be altered by the Char2 value.
   - Char2: For KeyType 12H (Extended Shift), the value to OR into the byte when the make code is seen and who's inverted value is ANDed when the break code is seen. For KeyType 13H (Extended Toggle), the value XORed into the byte on each make code seen (break code ignored).
   - Char3: Use in place of the Char2 when the "secondary" key prefix (hex E0) is seen immediately prior to this key.

     Examples of usage are: Char1/2 can define single shift status bits to set/clear/toggle. Char2 can be a set of coded bits (delineated by Char1) that will be set to a numeric value when the key is hit and cleared to zero when released (or on the next hit if toggle). The whole byte can be set to Char2 when Char1 has all bits on.
10. Space Key Note: The key treated as the space character should have a flag set in its AccentFlags field for each possible accent (that is, for each defined AccentEntry in the AccentTable). And each AccentEntry should have the SPACE character defined as one of its accented characters, with the translation being to the same value as the accent character itself. The reason for this is that, by definition, an Accent Key followed by the space character maps to the accent character alone. If the table is not set up as just described, a not-an-accent beep will be sounded whenever the accent key followed by a space is pressed.

    Note that the space key is defined as a SpecKeyA (type 4) because its use in conjunction with the Alt key is allowed. In that case it will still return the ASCII space character. It will also return the ASCII space character when used with the Ctrl key.

    This works correctly except in the case of the diaresis accent (double-dot) in code page 437. Here, the space is treated as an invalid character and the beep result occurs, with the diaresis represented by double quote. Characters are displayed depending upon the language in effect when the invalid diaresis is encountered. For some languages the character substituted is the double-quote; for others, the character used is the F9h character.
11. KbdType will identify the hardware specific keyboard this table is for. The values and allowable types are the same as specified in the Get Keyboard Type, IOCtl 77H.
12. The DefaultTable flag in XtableFlags1 is used by the KEYB utility in loading code pages when changing from one language to another. It identifies the default code page to KEYB should KEYB not find one or both CODEPAGE= defined code pages.
13. The language IDs and sub-country IDs used are as follows:

## Category 4 – Function 50H

| Keyboard Layout Country Code | Keyboard Layout SubCountry Code | Country |
|---|---|---|
| US | 103 | United States |
| UK | 168 | United Kingdom |
| UK | 166 | United Kingdom |
| GR | 129 | Germany |
| FR | 120 | France |
| FR | 189 | France |
| IT | 142 | Italy |
| IT | 141 | Italy |
| SP | 172 | Spain |
| DK | 159 | Denmark |
| NL | 143 | Netherlands |
| SU | 153 | Finland |
| NO | 155 | Norway |
| PO | 163 | Portugal |
| SV | 153 | Sweden |
| SF | 150F | Swiss-French |
| SG | 150G | Swiss-German |
| CF | 058 | Canadian-French |
| BE | 120 | Belgium |
| LA | 171 | Latin-American Spanish |

14. Keytype 15: When Alt or Alt+Shift key is pressed both the XlatedChar and XlatedScan in the CharData record will have the same value.
15. If the Charx value is in the range of 1 - 7 then Charx identifies an accent key; otherwise Charx is treated as a valid ASCII character. This does not apply to Ctl-Charx sequences.
16. If either the AltGrf, Alt Shift, or Alt Ctl are pressed and Char3 is zero, the Alt key will be used to translate to a valid result.
17. The keypad "*" key on the Enhanced keyboard, while producing the same scan code/character as that of the regular Personal Computer AT keyboard, is treated a bit differently, since a dedicated Print Screen key exists on the Enhanced keyboard. The following scan codes/characters (respectively) are returned by the base keyboard device driver for the Enhanced keyboard "*" key on the keypad:
    UNSHIFTED: 37H/2AH
    SHIFTED: 37H/2AH
    Ctrl: 96H/00
    Alt: 37H/00
18. Size: The code page described here has the following dimensions:

```
Xlate Header                = 40
127 KeyDefs @ 7 Bytes       = 889
7 AccentEntries @ 46 Bytes  = 322
                              ---
                            1251 Bytes
```

## Returns

None

## Remarks

This request changes the device driver resident code page for the system. This IOCtl updates the ZERO entry of the code page control block.

# Category 4 – Function 51H

Set Input Mode

## Parameter Packet Format

| Field | Length |
|---|---|
| Mode | BYTE |

## Data Packet Format

None

## Where

***Mode***

is a 1-byte field containing:

| *Bit* | *Meaning* |
|---|---|
| 1xxxxxx1 | Shift Report |
| 0xxxxxx0 | ASCII mode |
| 1xxxxxxx | BINARY mode |

## Returns

None

## Remarks

This request is used to pass the current input mode to the keyboard device driver. The keyboard device driver will maintain the mode separately for each session. The caller can interrogate the mode using function code 71H. The mode is also returned on every READ CHARACTER DATA RECORD(S) call, function code 74H, and PEEK CHARACTER DATA RECORD call, function code 75H. The default input mode is ASCII. The device driver uses the mode when processing CTL functions and reporting the shift state when shift report is set on.

Set Interim Character Flags

## Parameter Packet Format

| Field | Length |
|---|---|
| Flag | BYTE |

## Data Packet Format

None

## Where

***Flag***

is a 17-byte field containing flag bits. A bit set = to 1 indicates the state listed below:

| ***Bit*** | ***Meaning*** |
|---|---|
| 7 | Interim console flag on |
| 6 | Reserved = 0 |
| 5 | Program requested on-the-spot conversion |
| 4 | Reserved = 0 |
| 3 | Reserved = 0 |
| 2 | Reserved = 0 |
| 1 | Reserved = 0 |
| 0 | Reserved = 0 |

## Returns

None

## Remarks

This request is used to set the interim character flags maintained by the keyboard device driver. The keyboard device driver will maintain the flags separately for each session. The keyboard device driver passes the interim character flags with each character data record to keyboard monitors.

# Category 4 –
# Function 53H

Set Shift State

## Parameter Packet Format

| Field | Length |
|---|---|
| Shift State | WORD |
| NLS | BYTE |

## Data Packet Format

None

## Where

***Shift State***

is a word field containing shift states.

Word High Byte

| *Bit* | *Meaning* |
|---|---|
| 15 | SysReq Key down |
| 14 | Caps Lock Key down |
| 13 | NumLock Key down |
| 12 | ScrollLock Key down |
| 11 | Right Alt Key Down |
| 10 | Right Ctrl Key Down |
| 9 | Left Alt Key Down |
| 8 | Left Ctrl Key Down |

Word Low Byte

| *Bit* | *Meaning* |
|---|---|
| 7 | Insert On |
| 6 | Caps Lock On |
| 5 | NumLock On |
| 4 | ScrollLock On |
| 3 | Either Alt Key Down |
| 2 | Either Ctrl Key Down |
| 1 | Left Shift Key Down |
| 0 | Right Shift Key Down |

***NLS***

is a byte field containing NLS shift status.

## Returns

None

## Remarks

This request is used to set the current shift state for the keyboard. The keyboard device driver maintains the shift state separately for each logical keyboard. Note that this call will override the shift state set by previous shift keystrokes. Also the shift state set by this function code will be overridden by any subsequent shift keystrokes. The shift state is inserted into the character data record that is built for each incoming keystroke.

Set Typematic Rate and Delay

## Parameter Packet Format

| Field | Length |
|---|---|
| Delay | WORD |
| Rate | WORD |

## Data Packet Format

None

## Where

***Delay***

specifies the typematic delay in milliseconds. A value greater than the maximum value defaults to the maximum value.

***Rate***

specifies the typematic rate in characters per second. A value greater than the maximum value defaults to the maximum value.

## Returns

None

## Remarks

This request is used to set the keyboard typematic rate and delay to the values specified in the request.

# Category 4 –
# Function 55H

Notify of Change of Foreground Session

## Parameter Packet Format

| Field | Length |
|---|---|
| Session Number | WORD |
| Terminate Flag | WORD |

## Data Packet Format

None

## Where

***Session Number***

is a one word field containing the new foreground session. The session number must fall within the range 0 through 15.

***Terminate Flag***

is a one word field indicating whether the session is being terminated. A -1 value indicates termination, otherwise non-termination is indicated.

## Returns

None

## Remarks

This request is used to tell the keyboard device driver a new foreground session has been made active. The keyboard device driver will set the shift state of the keyboard to the state that was current when the new session was last active. The keyboard device driver will begin using the keyboard input buffer (KIB) and keystroke monitor chain associated with the new session.

This command is restricted and may only be used by the first process that makes the call. This function is for the use of the System Session Manager or the Presentation Manager.

Set Session Manager Hot Key

## Parameter Packet Format

| Field | Length |
|---|---|
| State | WORD |
| Make Code | BYTE |
| Break Code | BYTE |
| Hot Key ID | WORD |

## Data Packet Format

None

## Where

***State***

High Byte

| *Bit* | *Meaning* |
|---|---|
| 15 | SysReq |
| 14 | Caps Lock Key down |
| 13 | NumLock Key down |
| 12 | ScrollLock Key down |
| 11 | Right Alt Key Down |
| 10 | Right Ctrl Key Down |
| 9 | Left Alt Key Down |
| 8 | Left Ctrl Key Down |

Low Byte

| *Bit* | *Meaning* |
|---|---|
| 7 | Reserved = 0 |
| 6 | Reserved = 0 |
| 5 | Reserved = 0 |
| 4 | Reserved = 0 |
| 3 | Reserved = 0 |
| 2 | Reserved = 0 |
| 1 | Left Shift Key Down |
| 0 | Right Shift Key Down |

***Make Code***
is the scan code of the hot key make

***Break Code***
is the scan code of the hot key break

***Hot Key ID***
is the hot key ID (Value is set by the caller).

## Returns

None

## Category 4 –
## Function 56H

### Remarks

This request is used by the session manager to set a list of keyboard hot keys the keyboard device driver will begin looking for. The new hot key is global, that is, it applies to all sessions. Up to 16 hot keys can be defined by the session manager for handling by the keyboard device driver. This IOCtl call will only be successful if done by the process which initially invoked IOCtl 55H (Set Foreground Session). The combination of the shift flags in the first word and the scan codes in the second allow the session manager to set hot key combinations such as Alt+Esc. The hot key is triggered on detection of the scan code for the hot key break.

**Note:** If a DOS mode application has claimed hardware interrupt 9 or interrupt 50, the hot key will be triggered on detection of the break scan code for the required shift key.

A hot key can be redefined by calling this function with the same hot key ID.

Set KCB

## Parameter Packet Format

| Field | Length |
|---|---|
| KCB Handle | WORD |

## Data Packet Format

None

## Where

***KCB Handle***

is the handle identifying the logical keyboard's KCB.

## Returns

None

## Remarks

This request binds the specified logical keyboard (KCB) to the physical keyboard for this session.

This function will return an UNKNOWN COMMAND error if the caller is in the Presentation Manager session.

# Category 4 –
# Function 58H

Set Code Page ID

## Parameter Packet Format

| Field | Length |
|---|---|
| Code Page Translation Table Pointer | DWORD |
| Code Page ID | WORD |
| Set to Zero | WORD |

## Data Packet Format

None

## Where

***Code Page Translation Table Pointer***

is the selector:offset pointing to the code page translation table.

***Code Page ID***

is one word containing the current code page ID.

## Returns

None

## Remarks

Sets the code page translation table used by the current KCB to the code page translation table identified by the input parameter. This IOCtl is callable from the DOS mode.

Set Read/Peek Notification

## Parameter Packet Format

| Field | Length |
|---|---|
| Session Number | WORD |
| Read/Peek Notification Flags | WORD |

## Data Packet Format

None

## Where

***Session Number***

indicates the session to be enabled or disabled.

***Read/Peek Notification Flags***

contains the following flags:

| Bit Number | Meaning |
|---|---|
| 15-1 | Reserved, set to 0. |
| 0 | Mask indicating whether to enable or disable read/peek notification, where:<br>0 = Disable read/peek notification.<br>1 = Enable read/peek notification. |

## Returns

None

## Remarks

Sets the DDFlags of a keypacket to instruct the keyboard device driver to send a special monitor packet down the monitor chain. This packet is sent on every read request seen by the keyboard device driver for a particular screen group. This continues until the screen group is terminated or a subsequent call disables Read Notification.

The process ID (PID) of the first caller of this IOCtl in the system will be saved by the keyboard device driver. Subsequent invocations of this IOCtl by a different PID will result in UNKNOWN COMMAND IOCtl error being returned.

# Category 4 –
# Function 5AH

Alter Keyboard LEDs.

## Parameter Packet Format

| Field | Length |
|---|---|
| State of each LED | WORD |

## Data Packet Format

None

## Where

***State of each LED***

is a bit mask indicating the new LED indicators state. A set bit means that a particular LED will be turned on. A clear bit indicates the LED will be turned off. The bit mask is defined as follows:

| Bit # | Meaning |
|---|---|
| 15-3 | Reserved, set to 0. |
| 2 | If set, Caps Lock LED turned on.<br>If clear, Caps Lock LED turned off. |
| 1 | If set, Num Lock LED turned on.<br>If clear, Num Lock LED turned off. |
| 0 | If set, Scroll Lock LED turned on.<br>If clear, Scroll Lock LED turned off. |

If any of the reserved bits are set, an INVALID PARAMETER error is returned.

## Returns

None.

## Remarks

Alters the Keyboard LEDs without changing the operational state of the keyboard's mode of scan code generation.

This IOCtl is reserved for the Presentation Manager session system use. It cannot be called by applications. Requests from applications will return an UNKNOWN COMMAND error.

Because it is not possible to separate the keyboard's LED and operational state for the 88/89 Key Enhanced Keyboard, the Keyboard Device Driver will perform a functional no-op for IOCtl requests received when the 88/89 Key Enhanced Keyboard is the attached system keyboard.

Set NLS and Custom Code Page

## Parameter Packet Format

| Field | Length |
|---|---|
| Code Page Translation Table Pointer | DWORD |
| Code Page Number | WORD |
| Code Page to Load | WORD |
| Hot Key ID | WORD |

## Data Packet Format

None

## Where

***Code Page Translation Table Pointer***

is the selector:offset pointing to the code page translation table.

**Note:** The keyboard device driver does not do machine model or submodel determination to validate this translate table address for a given machine or keyboard. This determination is the responsibility of the keyboard subsystem and system initialization routines.

***Code Page Number***

is one word identifying the code page number.

***Code Page to Load***

is one word identifying the number of the code page to load, 1 or 2. A -1 indicates a custom code page for which the segment containing the custom code page will be locked. This option is not valid for the DOS mode.

***Hot Key ID***

is the hot key ID (Value is set by the caller).

## Returns

None

## Remarks

This request is used to install one of two possible code page translation tables into the device driver. This IOCtl will update the number one or two entry of the code page control block; entry zero is the device driver resident code page translation table. Note that this IOCtl is similar to IOCtl 50H, Set Code Page, except that different entries in the code page control block are updated. This IOCtl is callable from the DOS mode.

# Category 4 –
# Function 5DH

Create Keyboard creates a new logical keyboard.

## Parameter Packet Format

| Field | Length |
|---|---|
| KCB ID | WORD |
| CodePageID | WORD |

## Data Packet Format

None

## Where

***KCB ID***

is one word containing a unique value (the DosOpenHandle) used to identify the new logical keyboard. A zero indicates the default keyboard.

***CodePageID***

is a word containing the desired initial code page to be used for the new logical keyboard.

## Returns

None

## Remarks

The desired initial code page should be either the default code page for the system or one of the two optional code pages that may be prepared in the CODEPAGE= statement in CONFIG.SYS.

This function returns an UNKNOWN COMMAND error if the caller is in the Presentation Manager Session.

Destroy Keyboard destroys an existing logical keyboard.

## Parameter Packet Format

| Field | Length |
|---|---|
| KCB ID | WORD |

## Data Packet Format

None

## Where

***KCB ID***

is one word containing a unique value used to identify the logical keyboard. A zero indicates the default keyboard.

## Returns

None

## Remarks

This function returns an UNKNOWN COMMAND error if the caller is in the Presentation Manager Session.

# Category 4 – Function 71H

Get Input Mode

## Parameter Packet Format

None

## Data Packet Format

| Field | Length |
|---|---|
| Mode | BYTE |

## Where

***Mode***

is a 1-byte field containing one of the following values:

| *Bit* | *Meaning* |
|---|---|
| 1xxxxxx1 | Shift Report |
| 0xxxxxx0 | ASCII mode |
| 1xxxxxxx | BINARY mode |

## Returns

None

## Remarks

This request is used to obtain the input mode of the session of the currently active process. The input mode can be set with function code 51H. The input mode is meaningful for Ctrl-C, Ctrl-P, Ctrl-S, Ctrl-Break, Ctrl-ScrollLock, and Ctrl-PrtSc processing only.

Get Interim Character Flags

## Parameter Packet Format

None

## Data Packet Format

| Field | Length |
|---|---|
| Flags | BYTE |

## Where

***Flags***

is a 1-byte field containing flag bits. A bit set = to 1 indicates the state listed below:

| *Bit* | *Meaning* |
|---|---|
| 7 | Interim console flag on |
| 6 | Reserved = 0 |
| 5 | Program requested on-the-spot conversion |
| 4 | Reserved = 0 |
| 3 | Reserved = 0 |
| 2 | Reserved = 0 |
| 1 | Reserved = 0 |
| 0 | Reserved = 0 |

## Returns

None

## Remarks

This request is used to obtain the interim character flags maintained by the keyboard device driver.

# Category 4 –
# Function 73H

Get Shift State

## Parameter Packet Format

None

## Data Packet Format

None

## Returns

None

## Remarks

This request is used to obtain the shift state of the session of the currently active process. The shift state is set by incoming key strokes and by function code 53H calls.

Refer to function 53H, Set Shift State for the data structure.

Read Character Data Record(s)

## Parameter Packet Format

| Field | Length |
|---|---|
| Transfer Count | WORD |

## Data Packet Format

See the "KbdCharIn – Read Character, Scan Code" API function call for the CharData data structure.

## Where

***Transfer Count***

is a one word field containing the record transfer count. The sign bit of this word is set to request one of the following actions:

***0*** Wait for the requested number of key strokes to become available. The device driver will block the requestor until all requested character data records are available and have been transferred to the caller.

***1*** Do not wait for the requested number of key strokes to become available. In this case, all characters currently available will be transferred, up to the requested transfer count.

## Returns

None

## Remarks

This request is used to obtain one or more character data records from the keyboard input buffer (KIB) for the session of the currently active process. Note that if shift report is on then the CharData record may not contain a character but a shift state change in the shift status field.

This function returns an UNKNOWN COMMAND error if the caller is in the Presentation Manager Session.

# Category 4 –
# Function 75H

Peek Character Data Record

## Parameter Packet Format

| Field | Length |
|---|---|
| Status | WORD |

## Data Packet Format

See the "KbdCharIn – Read Character, Scan Code" API function call for the CharData data structure.

## Where

***Status***

is a one word field which contains either 0 to indicate no key stroke is available or 1 to indicate that a character data record is being returned. The sign bit is set to either 0 if the current input mode is ASCII or 1 if the input mode is BINARY.

## Returns

None

## Remarks

This request is used to obtain one character data record from the head of the keyboard input buffer (KIB) of the session for the currently active process. The character data record is not removed from the KIB. Note that if shift report is on then the CharData record may not contain a character but a shift state change in the shift status field.

This function returns an UNKNOWN COMMAND error if the caller is in the Presentation Manager Session.

Get Session Manager Hot Key

## Parameter Packet Format

| Field | Length |
|---|---|
| Type | WORD |

## Data Packet Format

None

## Where

***Type***

is a one word subtype indicating the type of information to return:

***0*** Return the maximum number of hot keys the keyboard device driver can support.

***1*** Return the number of hot keys currently defined in the system and return the key information for each.

***Returned Data Buffer***

If parameter list on entry was 1, one or more hot key data structures as described under IOCtl function code 56H (Set Session Manager Hot Key).

## Returns

None

## Remarks

This request is used to obtain the scan code the keyboard device driver is using as the session manager hot key.

This function should first be called with parameter list subtype = 0 to determine the maximum number of hot keys the device driver can support. The value returned should be used to determine the required size of the data buffer on a subsequent call to return the hot key data structures (parameter list subtype = 1).

# Category 4 – Function 77H

Get Keyboard Type

## Parameter Packet Format

None

## Data Packet Format

| Field | Length |
|---|---|
| Keyboard Type | WORD |
| Reserved = 0 | DWORD |

## Where

***Keyboard Type***

is a 1-word field containing:

High Byte: Reserved = 0

Low Byte:

| *Value* | *Meaning* |
|---|---|
| 00H | Personal Computer AT keyboard |
| 01H | Enhanced Keyboard |
| 02H to FFH | Reserved = 0 |

## Returns

None

## Remarks

This request returns the keyboard type.

For specific keyboard hardware IDs, see Category 4, IOCtl Function 7AH.

Get Code Page ID

## Parameter Packet Format

| Field | Length |
|---|---|
| Code Page ID | WORD |

## Data Packet Format

None

## Where

***Code Page ID***

is one word containing the current code page ID in use. Values other than Code Page IDs that may be returned are as follows:

0 indicates that PC US 437 is being used.

-1 indicates that a custom code page is installed.

## Returns

None

## Remarks

This request returns the code page in use by the current KCB. This IOCtl is callable from the DOS mode.

# Category 4 – Function 79H

Translate Scan Code to ASCII

## Parameter Packet Format

| Field | Length |
|---|---|
| Code Page ID | WORD |

## Data Packet Format

| Field | Length |
|---|---|
| CharData Record | 10 BYTES |
| KbdDDFlags | WORD |
| Xlate Flags | WORD |
| Xlate State1 | WORD |
| Xlate State2 | WORD |

## Where

***Code Page ID***
is one word containing the current code page ID.

***CharData Record***
See the "KbdCharIn – Read Character, Scan Code" API function call for the CharData data structure.

***KbdDDFlags***
as defined in the Device Monitor packets in the section in this book on the keyboard device driver.

***Xlate Flags***

High Byte

| *Value* | *Meaning* |
|---|---|
| 8-15 | Reserved = 0 |

Low Byte

| *Value* | *Meaning* |
|---|---|
| 1-7 | Reserved = 0 |
| 0 | Translation complete |

***Xlate State1 and Xlate State2***
identifies the state of translation across successive calls. Initially these words should be 0. They should be reset to 0 when the caller wants a new start to translation. Note that it may take several calls to this IOCtl to complete a character so these fields should not be touched unless a fresh start to translation is desired. These fields are set to 0 at the completion of translation.

## Returns

None

## Remarks

This request translates a scan code in a CharData record to an ASCII character. Optionally a code page may be specified to use for translation otherwise the code page of the active KCB will be used. To translate a given scan code with a particular shift state, indicate the shift state desired by way of the Shift State Field of the CharDataRec.

# Category 4 – Function 7AH

Get the hardware ID for the currently attached keyboard.

## Parameter Packet Format

None

## Data Packet Format

| Field | Length |
|---|---|
| Data Length | WORD |
| Hardware ID | WORD |

## Where

***Data Length***

On input, Data Length specifies the maximum number of returned data bytes being requested by the caller. The length value on input includes the length field's size. An input length field value of less than 2 is returned to the caller with an INVALID PARAMETER error.

On output, Data Length always indicates the maximum number of bytes that this function can return. For OS/2 Version 1.2, this return length size is 4 bytes.

***Hardware ID***

is the attached hardware ID.

## Returns

Returns the hardware ID for the currently attached keyboard.

## Remarks

In the past, all keyboards could be supported by knowing the hardware family information available via keyboard IOCtl 77H. However, in OS/2 Version 1.2, the addition of the 122 key keyboard recognition was not containable by hardware family information alone. The 122 key keyboard has a number of differences from the other 88/89 Key Enhanced Family keyboards. Therefore, applications performing keystroke specific functions may need to determine specifically which keyboard is attached.

OS/2 supported keyboards and their hardware generated IDs are as follows:

| Keyboard | Hardware ID |
|---|---|
| PC-AT Standard Keyboard | 0001H |
| 101 Key Enhanced Keyboard | AB41H |
| 102 Key Enhanced Keyboard | AB41H |
| 88 Key Enhanced Keyboard | AB54H |
| 89 Key Enhanced Keyboard | AB54H |
| 122 Key Enhanced Keyboard | AB85H |

Get the keyboard's current code page support information.

## Parameter Packet Format

None

## Data Packet Format

| Field | Length |
|---|---|
| Data Length | WORD |
| KbdCP | WORD |
| Ctry | ASCIIZ |
| SCtry | ASCIIZ |

## Where

***Data Length***

is the input return data length requested or the output data structure length required (in bytes).

On input, the length field value specifies the caller's return data area size. A minimum value of 2 is required so that the length value can be properly returned. An INVALID PARM error code is returned for input length values less than 2.

On output, the remaining returned data depends on the input length field value. For OS/2 Version 1.2, length field values of 12 bytes or greater will result in the full data structure being returned. For length requests less than 12, only the portion of returned data fields which can be completely returned, including ASCIIZ string terminators, is returned. The length field on return always represents the byte count of returned data, including string terminators.

***KbdCP***

is the active keyboard code page ID number.

***Ctry***

is the active keyboard country code in ASCIIZ string format.

***SCtry***

is the active keyboard sub-country code in ASCIIZ string format.

## Returns

None

## Remarks

This request returns the keyboard's current code page support information. This information is particularly useful for utilities like KEYB. The Code Page Support Information consists of the keyboard's active code page and the country and sub-country codes active for the keyboard layout. These items are the system-level support factors used by the keyboard device driver for global translation.

# Category 5 Printer Control IOCtl Commands

The following is a summary of Category 5 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 42H | Set frame control (CPL, LPI) |
| 44H | Set infinite retry |
| 45H | Reserved |
| 46H | Initialize printer |
| 48H | Activate font |
| 62H | Get frame control |
| 64H | Get infinite retry |
| 66H | Get printer status |
| 69H | Query active font |
| 6AH | Verify font |

Set Frame Control

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Characters per Line | BYTE |
| Lines per Inch | BYTE |

## Where

***Command Information***
is reserved and must be set to zero.

***Characters Per Line***
Valid numbers are 80 and 132.

***Lines Per Inch***
Valid numbers are 6 and 8.

## Returns

In the request packet status field:

| | |
|---|---|
| **0100H** | Completed successfully. |
| **8103H** | Invalid command if CPL/LPI not (80,6), (80,8), (132,6) or (132,8). |
| **810AH** | Write fault if monitor registered and dh_MonWrite fails. |
| **8102H** | Device not ready if monitors not registered. |
| **810AH** | Write fault if monitors not registered. |

## Remarks

None

# Category 5 – Function 44H

Set Infinite Retry

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Data | BYTE |

## Where

***Command Information***

is reserved and must be set to zero.

***Data***

The data is defined as:

0 = Disable Infinite retry
1 = Enable Infinite retry

## Returns

In the request packet status field:

**0100H** Completed successfully.
**8103H** Invalid command if data not 0 or 1.

## Remarks

If a monitor is registered for the printer device driver, infinite retry is always enabled. When a monitor is registered a disable request will return a good return code and the function will not be performed.

Initialize Printer

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To Zero | BYTE |

## Where

***Command Information***
is reserved and must be set to zero.

## Returns

in the request packet status field:

**0100H** Completed successfully.
**810AH** Write fault if monitor registered and dh_MonWrite fails.

## Remarks

None

# Category 5 – Function 48H

Activate Font

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Code Page | WORD |
| Font ID | WORD |

## Where

***Command Information***

is reserved and must be set to zero

***Code Page***

is the value of the code page to make the currently active code page.

| | |
|---|---|
| 0000H | If the Code Page value and Font ID are specified as zero (0), set printer to hardware default code page and font. |
| 0001H-FFFFH | Valid code page numbers. |

***Font ID***

is the ID value of the Font to make currently active.

| | |
|---|---|
| 0000H | If the Code Page value and Font ID are specified as zero (0), set printer to hardware default code page and font.<br><br>If Font ID is zero and the code page is a valid non-zero, then any font within the specified code page is acceptable. |
| 0001H-FFFFH | Valid Font ID numbers, font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label and are also entered in the DEVINFO statement for the printer. |

## Returns

in the request packet status field:

| | |
|---|---|
| **0100H** | Completed successfully. |
| **810AH** | Write fault if monitor registered and dh_MonWrite fails. |
| **C102H** | Code page is not available. |
| **C103H** | No code page function because spooler not started. |
| **C104H** | Font ID is not available (verify). |
| **C109H** | Error caused by switcher error not by input parameters. |
| **C10AH** | Error caused by invalid printer name as input. |
| **C10DH** | Received code page request when code page switcher not initialized. |
| **C10FH** | SFN table full, cannot activate another entry. |
| **C113H** | DASD error reading font file. |
| **C115H** | DASD error reading font file definition block. |
| **C117H** | DASD error while writing to temporary spool file. |
| **C118H** | Disk full error while writing to temporary spool file. |
| **C119H** | Spool file handle was bad. |

## Remarks

None

# Category 5 – Function 62H

Return Frame Control

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Characters per Line | BYTE |
| Lines per Inch | BYTE |

## Where

***Command Information***
Is reserved and must be set to zero.

***Characters Per Line***
On return, field is set to 80 or 132.

***Lines Per Inch***
On return, field is set to 6 or 8.

## Returns

In the request packet status field:

**0100H** Completed successfully.

## Remarks

None

Return Infinite Retry

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Data | BYTE |

## Where

***Command Information***
is reserved and must be set to zero.

***Data***
On return, Data byte is set:

0 = Infinite retry is disabled.
1 = Infinite retry is enabled.

## Returns

in the request packet status field:

**0100H** Completed successfully.

## Remarks

None

# Category 5 – Function 66H

Return Printer Status

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Data | BYTE |

## Where

***Command Information***
is reserved and must be set to zero.

***Data***
On return, Data byte is set:

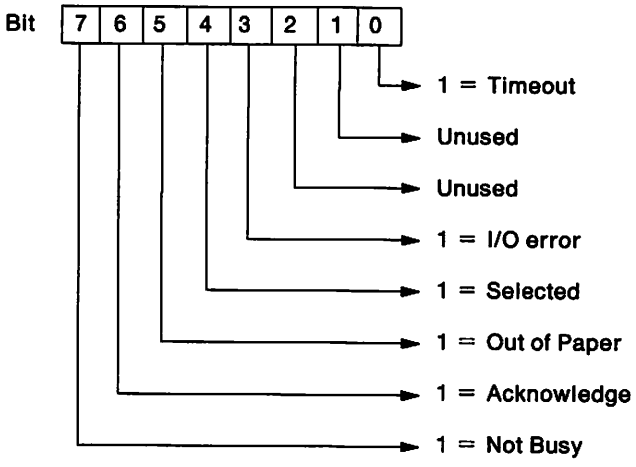



Figure 7-1. Data Byte

## Returns

in the request packet status field:

**0100H** Completed successfully.

## Remarks

None

Query Active Font

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Code Page | WORD |
| Font ID | WORD |

## Where

***Command Information***
is reserved and must be set to zero.

***Code Page***
On return, is set to currently active code page.

| | |
|---|---|
| 0000H | If the Code Page value and Font ID are returned as zero (0), the printer is set to the hardware default code page and font. |
| 0001H-FFFFH | Valid code page numbers. |

***Font ID***
On return, is the ID value of the Font which is currently active.

| | |
|---|---|
| 0000H | If the Code Page value and Font ID are specified as zero (0), the printer is set to the hardware default code page and font.<br><br>If Font ID is zero and code page is non-zero, no error will be returned if any Font ID is available for the specified code page. |
| 0001H-FFFFH | Valid Font ID numbers, font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label and are also entered in the DEVINFO statement for the printer. |

## Returns

in the request packet status field:

| | |
|---|---|
| **0100H** | Completed successfully. |
| **C103H** | No code page function because spooler not started. |
| **C109H** | Error caused by switcher error, not by input parameters. |
| **C10AH** | Error caused by invalid printer name as input. |
| **C10DH** | Received code page request when code page switcher not initialized. |
| **C110H** | Received request for SFN not in SFN table. |

## Remarks

None

# Category 5 –
# Function 6AH

Verify that a particular code page and font is available for the specified printer

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Code Page | WORD |
| Font ID | WORD |

## Where

***Command Information***
is reserved and must be set to zero.

***Code Page***
The Code Page number to validate.

Values may be 0 to 65535.

***Font ID***
The Font ID value to validate.

Values may be 0 to 65535. The Font ID is contained in the font file for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label and are also entered in the DEVINFO statement for the printer.

**Note:** A value of zero (0) for both the Code Page number and Font ID indicates the default hardware code page and font; this combination is always valid.

## Returns

in the request packet status field:

**C100H** Completed successfully
**C102H** Code page is not available.
**C103H** No code page function because spooler not started.
**C104H** Font ID is not available (verify).
**C10AH** Error caused by invalid printer name as input.
**C10DH** Received code page request when code page switcher not initialized.

## Remarks

None

# Category 7 Mouse Control IOCtl Commands

The following is a summary of Category 7 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 50H | Notification of Session Switch Completion |
| 51H | Notification of Display Mode Change |
| 52H | Notification of Impending Session Switch |
| 53H | Reassigns the Current Mouse Scaling Factors |
| 54H | Assigns a New Mouse Event Mask |
| 55H | Reserved |
| 56H | Sets the Pointer Shape |
| 57H | Frees the Mouse to Draw the Pointer anywhere on the Screen (unmark collision area) |
| 58H | Restricts the Mouse from Pointer Drawing in Specified Area(s) of the Screen (mark collision area) |
| 59H | Specifies/Replaces the Pointer Position |
| 5AH | Specifies the Pointer Draw Device Driver Address (OS/2 mode only) |
| 5BH | Specifies the Pointer Draw Device Driver Address (DOS mode only) |
| 5CH | Sets a Subset of the Current Mouse Device Driver Status Flags |
| 5DH | Notification of mode switch completion |
| 60H | Gets the Number of Mouse Buttons Supported by the Device Driver |
| 61H | Gets the Mouse Device's Motion Sensitivity |
| 62H | Gets the Current Mouse Device Driver Status Flags |
| 63H | Reads the Mouse Event Queue |
| 64H | Gets the Current Event Queue Status |
| 65H | Gets the Current Mouse Event Mask |
| 66H | Gets the Current Mouse Scaling Factors |
| 67H | Gets the Current Pointer Screen Position |
| 68H | Gets the Current Pointer Shape |
| 69H | Reserved |
| 6AH | Return the mouse device driver level/version. |

# Category 7 –
# Function 50H

Notification of session switch completion.

## Parameter Packet Format

None

## Data Packet Format

None

## Returns

None

## Remarks

This function notifies the mouse device driver that a session switch has been completed and the pointer may now be drawn. It is intended for use by the session manager.

Notification of Display Mode Change

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the Mode Data Definition Record. The Mode Data Definition Record has the following format:

| Field | Length |
|---|---|
| Length | WORD |
| Type | BYTE |
| Color | BYTE |
| Text Columns | WORD |
| Text Rows | WORD |
| Horizontal Resolution | WORD |
| Vertical Resolution | WORD |
| Attribute Format | BYTE |
| Buffer Address | DWORD |
| Buffer Length | DWORD |
| Full Buffer Size | DWORD |
| Partial Buffer Size | DWORD |
| Address of Extended Data Area | DWORD |

## Where

***Length***

is an input parameter to VioSetMode. Length specifies the length of the data structure in bytes including Length itself. The minimum structure size required is 3 bytes, and the maximum structure size required is 34 bytes. OS/2 will set to the first mode (in the list of modes supported by this display configuration) with a data structure matching the mode data specified.

***Type***

is a bit mask that contain specifications for the mode being set. The definitions of the bits follow:

```
xxxxxxxb b = 0 monochrome compatible mode
         b = 1 other

xxxxxxbx b = 0 text mode
         b = 1 graphics mode
xxxxxbxx b = 0 enable color burst
         b = 1 disable color burst
xxxxbxxx b = 0 VGA-compatible modes 0 - 13H
         b = 1 native mode
bbbbxxxx b = reserved, must be 0
```

***Color***

defines the number of colors as a power of 2. This is equivalent to the number of color bits which define the color. For example,

```
Color = 1 specifies 2 colors
Color = 2 specifies 4 colors
Color = 4 specifies 16 colors
Color = 8 specifies 256 colors

Color = 0 should be specified for
                monochrome modes 7, 7+, and F.
```

# Category 7 – Function 51H

***Text Columns***
are the number of text columns.

***Text Rows***
are the number of text rows. 25 rows are supported for the color graphics adapter. 25 and 43 rows are supported for the VGA adapter; and 25 and 50 rows are supported for the IBM Personal Systems/2 Display Adapter.

***Horizontal resolution***
is the number of pel columns.

***Vertical resolution***
is the number of pel rows.

***Attribute format***
identifies the format of the attributes.

***Number of Attributes***
identifies the number of attributes in a character cell.

***Buffer Address***
is the 32-bit physical address of the physical display buffer for this mode (returned value).

***Buffer Length***
is the length of the physical display buffer for this mode (returned value).

***Full Buffer Size***
is the size of the buffer required for a full save of the physical display buffer for this mode (returned value).

***Partial Buffer Size***
is the size of the buffer required for a partial (popup) save of the physical display buffer for this mode (returned value).

***Address of extended data area***
is the FAR address to an extended mode data structure, or zero if none. The format of the extended mode data structure is determined by the device driver and is unknown to OS/2.

## Data Packet Format

The Data Packet is a location in application storage that contains the configuration data for the display on which the mode is being set. The configuration data has the following format:

| Field | Length |
|---|---|
| Length | WORD |
| Adapter type | WORD |
| Display type | WORD |
| Memory | DWORD |
| Configuration # | WORD |
| Device driver version # | WORD |
| Flag bits | WORD |
| Hardware state buffer size | DWORD |
| Maximum buffer size - full save | DWORD |
| Maximum buffer size - partial save | DWORD |
| Offset to mode data | WORD |

## Where

***Length***
is an input parameter to VioGetConfig. Length specifies the length of the data structure in bytes, including the length itself.

The maximum size structure required in OS/2 Versions 1.0 and 1.1 is 10 bytes. The maximum size structure required in OS/2 Version 1.2 is variable and can be determined by issuing VioGetConfig with Length set to 2. When Length is set to 2 on input, VioGetConfig returns the maximum size structure required in the Length field on output. When Length is not equal to 2 on input, the Length field is unmodified on output.

***Adapter type***
is the display adapter type, with values set as follows:

0 = monochrome-compatible
1 = color graphics adapter
2 = enhanced graphics adapter
3 = VGA or PS/2 display adapter
4-6 = Reserved
7 = PS/2 display adapter 8514/A

**Note:** Values ranging from 0 - 4095 are reserved for IBM.

***Display type***
is the display/monitor type, with values indicating the following:

0 = monochrome display
1 = color display
2 = enhanced color display
3 = 8503 monochrome display
4 = 8512 or 8513 color display
5-8 = Reserved
9 = 8514 color display

**Note:** Values ranging from 0 - 4095 are reserved for IBM.

***Memory***
is the amount of memory on the adapter in bytes and is returned as a 32-bit value.

***Configuration #***
is the number of the display configuration which this data corresponds to.

***Device driver version #***
is the video device driver version number corresponding to this device driver.

***Flag bits***
are defined as follows:

0001H = power up display configuration
0002H = VGA pass-through

***Hardware state buffer size***
is the size of the buffer required by the video device driver to save the full hardware state excluding the physical display buffer.

***Maximum buffer size - full save***
is the maximum size buffer required by the video device driver to save the full physical display buffer.

***Maximum buffer size - partial save***
is the maximum size buffer required by the video device driver to save the portions of the physical display buffer that will be overlaid by a popup.

***Offset to mode data***
is the offset within the configuration data structure to the beginning of the mode data.

# Category 7 –
# Function 51H

## Returns

None

## Remarks

This call notifies the mouse device driver of a new display mode and display configuration.

When the Video Subsystem or registered Video Subsystem sets/resets the display mode, they must synchronize the mouse device driver pointer update routines by providing this notification record to the mouse device driver prior to switching display modes.

Notification of Impending Session Switch

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

| Field | Length |
|---|---|
| Session Number | WORD |
| Switch Notification Type | WORD |

## Data Packet Format

None

## Where

***Session Number***

is the session number for notification action. This value must be in the range of 0 < = session number < = MaxNumberOfScreenGroups. The Global InfoSeg defines the valid range session ID's.

***Switch Notification Type***

is the switch notification type. The values for this parameter follow:

| *Value* | *Meaning* |
|---|---|
| -1 | = The specified session number is terminating. |
| > = 0 | = The specified session number is being switched to. |

## Returns

None

## Remarks

This call notifies the mouse device driver that a session switch is about to take place.

This function sets a system pointer draw enable/disable flag which does not allow pointer drawing until the flag is cleared by a Category 7, Function 50H call.

# Category 7 –
# Function 53H

Reassigns the Current Mouse Scaling Factors

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

| Field | Length |
|---|---|
| Row Data | WORD |
| Column Data | WORD |

## Data Packet Format

None

## Where

***Row Data***

Row coordinate scaling factor.

***Column Data***

Column coordinate scaling factor.

The scaling factor values are positive integers in the range of:

0 < value < = (32K - 1)

## Returns

None

## Remarks

This function reassigns the current pointing device scaling factors.

Scaling factors are ratio values that determine how much relative movement is necessary before the mouse device driver will report a mouse event.

The ratios specify the number of mickeys per 8 pixels. The default ratio values are:

Vertical/Row ration - 16 mickeys per 8 pixels
Horizontal/Row ratio - 8 mickeys per 8 pixels.

Assigns a New Mouse Event Mask

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

| Field | Length |
|---|---|
| Event Mask | WORD |

## Data Packet Format

None

## Where

***Event Mask***

| *Bit* | *Meaning* |
|---|---|
| 7-15 | Reserved = 0 |
| 6 | Set if button 3 is down |
| 5 | Set if motion with button 3 down |
| 4 | Set if button 2 is down |
| 3 | Set if motion with button 2 down |
| 2 | Set if button 1 is down |
| 1 | Set if motion with button 1 down |
| 0 | Set if all mouse motion, no buttons |

A set bit is a value of 1.

## Returns

None

## Remarks

The mouse device driver gets the new mask for enabled/disabled mouse device events from the caller's parameter packet. This mask determines which events are to be queued, that is, readable by calling Category 7, Function 63H.

# Category 7 –
# Function 56H

Sets the Pointer Shape

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the Pointer Definition Record. The Pointer Definition Record has the following format:

| Field | Length |
|---|---|
| Buffer length | WORD |
| Columns | WORD |
| Rows | WORD |
| Column Hot Spot | WORD |
| Row Hot Spot | WORD |

## Data Packet Format

The Data Packet is a location in application storage that contains the pointer image buffer. The pointer image buffer contains a new pointer shape being defined by the user for the mouse device driver. The format of this buffer is dependent on the display mode. The buffer always consists of the AND pointer image data followed by the XOR pointer image data. The buffer always describes only one display plane.

## Where

***Buffer length***
is the length of pointer image buffer.

***Columns***
is the width in columns of pointer image.

***Rows***
is the height in rows of pointer image.

***Column Hot Spot***
is the column offset within pointer image to hotspot.

***Row Hot Spot***
is the row offset within pointer image to hotspot.

## Returns

None

## Remarks

In the Parameter Packet, the caller specifies a one-WORD height value (Rows) for the pointer shape and a one-WORD width value (Columns) for the pointer shape. For text and mono mode applications, these values must be equal to 1. For graphics mode applications, these values must be greater than or equal to 1.

The Length (in BYTES) of the pointer shape will vary depending on the display mode:

**Mono & Text**

```
Buffer length = (height in characters) *
                (width in characters) * 2 * 2
              = 1 * 1 * 2 * 2
```

**Note:** For text mode height and width must be 1, so length is always 4.

**Graphics**

```
Buffer length = (height in pels) *
                (width in pels) * (bits per pel) * 2 / 8
```

**Note:** Width (width in pels) must be a multiple of 8.

**Modes 4 & 5 (320 x 200)**

```
Buffer length = (height) * (width) * 2 * 2 / 8
```

**Mode 6 (640 x 200)**

```
Buffer length = (height) * (width) * 1 * 2 / 8
```

**Note:** Length calculations produce byte boundary buffer sizes.

All of the pointer definition record fields and the pointer shape buffer are validated using the session's mode table values. The parameter values must be the same orientation as the current session display mode:

- graphics = pixel values
- text = character values

# Category 7 – Function 57H

Frees the Mouse to Draw the Pointer anywhere on the Screen

## Parameter Packet Format

None

## Data Packet Format

None

## Returns

None

## Remarks

A collision area is a restricted area on the Screen that cannot be overwritten by pointer drawing. (See Category 7, Function 58H.)

This function frees the current collision area for a session so that the pointer can be drawn anywhere on the screen. If a collision area was declared for the session, the collision area is freed and the pointer position is checked. If the pointer was in the freed area, it is drawn. If the pointer was not in the freed area, no pointer manipulations take place.

Restricts the Mouse from Pointer Drawing in Specified Area(s) of the Screen

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains a data structure that specifies a restricted area on the screen (a collision area). The format of this data structure follows:

| Field | Length |
|---|---|
| Left Row Position | WORD |
| Left Column Position | WORD |
| Right Row Position | WORD |
| Right Column Position | WORD |

## Data Packet Format

None

## Returns

None

## Remarks

A collision area is a restricted area on the screen that cannot be overwritten by pointer drawing. Values contained in the data structure defining a collision area must be specified in either character or pixel values, depending on the current display mode.

If the collision area is defined as the entire screen, pointer drawing is disabled for the session.

The pointer will not be drawn in this area until a different area is specified through another call to this function or until Category 7, IOCtl function 57H is called.

# Category 7 –
# Function 59H

Specifies/Replaces the Pointer Position

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

| Field | Length |
|---|---|
| Row Position | WORD |
| Column Position | WORD |

## Data Packet Format

None

***Row Position***

is the new row coordinate pointer screen position.

***Column Position***

is the new column coordinate pointer screen position.

## Returns

None

## Remarks

The coordinate values are dependent on the display mode. Pixel values must be used if the display is in graphics mode. Character position values must be used if the display is in text mode.

This function does not override Category 7, Functions 57H and 58H; that is, it has no effect on current restricted areas of the Screen. If the pointer is directed into a restricted area (that is, a collision area), it remains invisible until moved out of the area or until the restrictions are removed.

Specifies the Address of the Pointer Draw Device Driver (OS/2 mode only)

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

| Field | Length |
|---|---|
| Pointer Entry | DWORD |
| Pointer DS | DWORD |

## Where

***Pointer Entry***

Contains two 1-word fields whose format is as follows:

Word 0 = Pointer Draw Rtn Device Driver's Entry Point Offset

Word 1 = Pointer Draw DD Entry Point Selector

***Pointer DS***

Contains two 1-word fields whose format is as follows:

Word 0 = Pointer Draw Rtn Device Driver's Data Segment selector

Word 1 = Reserved = 0.

## Data Packet Format

The Data Packet is a location in application storage that contains a the following data:

| Field | Length |
|---|---|
| Length of data | WORD |
| Display configuration # | WORD |
| Caller | WORD |

## Where

***Length of data***

is the length of the data structure (=6).

***Display configuration #***

is the display configuration number on which the pointer draw routine should draw.

***Caller***

specifies whether this call is made on behalf of an application or the Base Video Subsystem (BVS), where:

0 = application
1 = BVS

## Returns

None

# Category 7 – Function 5AH

## Remarks

This IOCtl is for the OS/2 mode only.

The call passes to the mouse device driver the selector:offset address of the entry point of the session's pointer draw routine for OS/2 mode display support. The pointer image draw routine is an installed pseudo character device driver. The mouse router/handler must:

- OPEN the pointer draw device driver.
- Query the pointer draw device driver for the address of its entry point.
- Pass the resulting address of the pointer draw entry point to the mouse device driver using the IOCtl described above.

The mouse device driver issues a far call to the pointer draw device driver when a mouse interrupt occurs that requires action concerning the pointer image.

In addition, the mouse device driver may call the pointer draw routine as a result of some action on the part of the application, such as:

- MouDrawPtr
- MouRemovePtr
- MouSetPtrPos
- MouSetPtrShape
- MouGetPtrShape
- MouSetDevStatus.

Specifies the Pointer Draw Device Driver Address (DOS mode only)

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

| Field | Length |
|---|---|
| Pointer Entry | DWORD |
| Pointer DS | DWORD |

## Where

***Pointer Entry***

Contains two 1-word fields whose format is:

Word 0 = Pointer Draw Rtn Device Driver's Entry Point Offset
Word 1 = Pointer Draw Rtn Device Driver's Entry Point Selector

***Pointer DS***

Entry contains two 1-word fields whose format is:

Word 0 = Pointer Draw Rtn Device Driver's Data Segment Selector
Word 1 = Reserved = 0

## Data Packet Format

The Data Packet is a location in application storage that contains the configuration data for the power-up display configuration. The configuration data has the following format:

| Field | Length |
|---|---|
| Length | WORD |
| Adapter type | WORD |
| Display type | WORD |
| Memory | DWORD |
| Configuration # | WORD |
| Device driver version # | WORD |
| Flag bits | WORD |
| Hardware state buffer size | DWORD |
| Maximum buffer size - full save | DWORD |
| Maximum buffer size - partial save | DWORD |
| Offset to mode data | WORD |

## Where

***Length***

is an input parameter to VioGetConfig. Length specifies the length of the data structure in bytes, including the length itself.

The maximum size structure required in OS/2 Versions 1.0 and 1.1 is 10 bytes. The maximum size structure required in OS/2 Version 1.2 is variable and can be determined by issuing VioGetConfig with Length set to 2. When Length is set to 2 on input, VioGetConfig returns the maximum size structure required in the Length field on output. When Length is not equal to 2 on input, the Length field is unmodified on output.

# Category 7 – Function 5BH

***Adapter type***
is the display adapter type, with values set as follows:

0 = monochrome-compatible
1 = color graphics adapter
2 = enhanced graphics adapter
3 = VGA or PS/2 display adapter
4-6 = Reserved
7 = PS/2 display adapter 8514/A

**Note:** Values ranging from 0 - 4095 are reserved for IBM.

***Display type***
is the display/monitor type, with values indicating the following:

0 = monochrome display
1 = color display
2 = enhanced color display
3 = 8503 monochrome display
4 = 8512 or 8513 color display
5-8 = Reserved
9 = 8514 color display

**Note:** Values ranging from 0 - 4095 are reserved for IBM.

***Memory***
is the amount of memory on the adapter in bytes and is returned as a 32-bit value.

***Configuration #***
is the number of the display configuration which this data corresponds to.

***Device driver version #***
is the video device driver version number corresponding to this device driver.

***Flag bits***
are defined as follows:

0001H = power up display configuration
0002H = VGA pass-through

***Hardware state buffer size***
is the size of the buffer required by the video device driver to save the full hardware state excluding the physical display buffer.

***Maximum buffer size - full save***
is the maximum size buffer required by the video device driver to save the full physical display buffer.

***Maximum buffer size - partial save***
is the maximum size buffer required by the video device driver to save the portions of the physical display buffer that will be overlaid by a popup.

***Offset to mode data***
is the offset within the configuration data structure to the beginning of the mode data.

## Returns
None

## Remarks
This IOCtl is for the DOS mode only. The parameter packet is the same as that required for the OS/2 IOCtl (see Category 7, Function 5AH).

The call passes to the mouse device driver the address of the entry point of a pointer draw routine for DOS mode display support. In DOS mode, pointer draw is only supported on the power-up display configuration.

The data is passed as selector : offset pairs. The DOS mode portion of the mouse device driver uses the VirtToPhys and PhysToVirt OS/2 calls to convert this address to the segment : offset real address for use in the DOS mode.

# Category 7 – Function 5CH

Sets a Subset of the Current Mouse Device Driver Status Flags

## Parameter Packet Format

The Parameter Packet is a location in application storage that contains the following data:

| Field | Length |
|---|---|
| Status Mask | WORD |

## Data Packet Format

None

## Where

***Status Mask***

has the following bit level definitions:

High Byte

| *Bit* | *Meaning* |
|---|---|
| 7-2 | Reserved = 0 |
| 1 | Set if mouse data returned in mickeys, not display units |
| 0 | Set if the interrupt level pointer draw routine is not called |

Low Byte

| *Bit* | *Meaning* |
|---|---|
| 7-0 | Reserved = 0 |

A set bit is a value of 1.

## Returns

None

## Remarks

This function is the complement to Category 7, Function 62H.

Only the high byte of the one-word mouse device driver status mask may be set.

If the data format flag is altered, the session's monitor chain and event queue are flushed.

It is useful to disable Interrupt time Pointer Draw Device Driver activity when an application is going to set into an unsupported display, for which, it intends to perform the pointer image drawing support. If this is desired, the application must set this mouse state prior to switching the display mode.

Informs the mouse that a display mode/configuration switch has been completed and that drawing operations may resume.

## Parameter Packet Format

Dummy FAR pointer (such as a double word of zeroes)

## Data Packet Format

Dummy FAR pointer (such as a double word of zeroes)

## Returns

None

## Remarks

This call does not provide return parameters or support input parameters.

If the pointer was hidden on the mode switch start, IOCtl 51H, then it is redrawn.

This function is the complement to the Category 7, IOCtl function 51H.

# Category 7 –
# Function 60H

Gets Number of Buttons Supported by the Mouse Device Driver

## Parameter Packet Format

None

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller the number of buttons supported by the driver. The format of the Data Packet is a follows:

| Field | Length |
|---|---|
| Number Supported | WORD |

## Where

***Number Supported***

is the number of buttons supported by the mouse device driver. Return values will be in the range of:

1 = one-button support
2 = two-button support
3 = three-button support

## Returns

This function returns to the caller the number of buttons supported by the mouse device driver.

## Remarks

None

Gets the Mouse Device's Motion Sensitivity

## Parameter Packet Format

None

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller the mouse device's motion sensitivity. The format of the Data Packet is as follows:

| Field | Length |
|---|---|
| Mickeys/Centimeter | WORD |

## Where

***Mickeys/Centimeter***

is the number of mickeys/centimeter supported by the mouse device. Return values will be in the range of:

0 < number of mickeys/centimeter < = (32K - 1)

## Returns

This function returns to the caller the motion sensitivity of the mouse device, as represented by the number of mickeys/centimeter.

## Remarks

None

# Category 7 –
# Function 62H

Gets the Current Mouse Device Driver Status Flags

## Parameter Packet Format

None

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller the current device status flags. The format of the Data Packet is as follows:

| Field | Length |
|---|---|
| Status Flags | WORD |

## Where

***Status Flags***

has the following bit level definitions:

High Byte

| *Bit* | *Meaning* |
|---|---|
| 7-2 | Reserved = 0 |
| 1 | Set if mouse data returned in mickeys |
| 0 | Set if the interrupt level pointer draw routine is not called |

Low Byte

| *Bit* | *Meaning* |
|---|---|
| 7-4 | Reserved = 0 |
| 3 | Set if pointer draw routine disabled by unsupported mode |
| 2 | Set if flush in progress |
| 1 | Set if block read in progress |
| 0 | Set if event queue busy with I/O |

A set bit is a value of 1.

## Returns

This function returns to the caller the current mouse device driver status flags.

## Remarks

This function is the complement to Category 7, Function 5CH.

Reads the Mouse Event Queue

## Parameter Packet Format

The Parameter Packet is a location in application storage where the caller indicates action to be taken if no event queue data is available. The format of the Parameter Packet is as follows:

| Field | Length |
|---|---|
| Read Type | WORD |

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller a ten-BYTE mouse event queue record. A mouse event queue record has the following format:

| Field | Length |
|---|---|
| Event Mask | WORD |
| Time | DWORD |
| Row Position | WORD |
| Column Position | WORD |

## Where

***Read Type***
is only used to determine the type of action to be taken if no event queue data is available. The values may be:

0 = Block the process (Wait) until event data is available

1 = Return a NULL record (No Wait) for the request.

***Event Mask***
(See Category 7, Function 65H)

***Time***
is the event time stamp in milliseconds.

***Row Position***
is the Pointer row coordinate.

***Column Position***
is the Pointer column coordinate.

## Returns

This function returns to the caller a mouse event from the mouse event (FIFO) queue.

## Remarks

None

# Category 7 – Function 64H

Gets the Current Event Queue Status

## Parameter Packet Format

None

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver will return to the caller the following data:

| Field | Length |
|---|---|
| Element Count | WORD |
| Queue Size | WORD |

## Where

***Element Count***

is the current number of event queue elements. The return value is a one-word value in the range of:

0 < = value < = MaxNumQueueElements.

***Queue Size***

is a one-word return value for the MaxNumQueueElements.

## Returns

This function returns to the caller the current number of queued elements in the mouse event queue and the maximum number of elements allowed in a mouse event queue.

## Remarks

MaxNumQueueElements is derived from the CONFIG.SYS QSIZE= keyword input parameter.

Gets the Current Mouse Event Mask

## Parameter Packet Format

None

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller a one-word Current Mouse Event Mask. The format of the Data Packet is as follows:

| Field | Length |
|---|---|
| Event Mask | WORD |

## Where

***Event Mask***

has the following bit level definitions:

| *Bit* | *Meaning* |
|---|---|
| 7-15 | Reserved = 0 |
| 6 | Set if button 3 is down |
| 5 | Set if motion with button 3 down |
| 4 | Set if button 2 is down |
| 3 | Set if motion with button 2 down |
| 2 | Set if button 1 is down |
| 1 | Set if motion with button 1 down |
| 0 | Set if all mouse motion, no buttons |

A set bit is a value of 1.

## Returns

This function returns to the caller the Current Mouse Event Mask.

## Remarks

The return values may be any valid combination of enabled/disabled event flags.

This is the complement function for Category 7, Function 54H.

# Category 7 – Function 66H

Gets the Current Mouse Scaling Factors

## Parameter Packet Format

None

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller the Current Mouse Scaling Factors. The format of the Data Packet is as follows:

| Field | Length |
|---|---|
| Row Data | WORD |
| Column Data | WORD |

## Where

***Row data***

is the row coordinate scaling factor.

***Column data***

is the column coordinate scaling factor.

The scaling factor values are positive integers in the range of:

0 < value < = (32K - 1)

## Returns

This function returns to the caller the Current Mouse Scaling Factors.

## Remarks

Scaling factors are ratio values that determine how much relative movement is necessary before the mouse device driver will report a mouse event.

The ratios specify the number of mickeys per 8 pixels. The default ratio values are:

Vertical/Row ration - 16 mickeys per 8 pixels
Horizontal/Row ratio - 8 mickeys per 8 pixels.

This is the complement function for Category 7, Function 53H.

Gets the Current Pointer Screen Position

## Parameter Packet Format

None

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver returns to the caller the Current Pointer Screen Position. The format of the Data Packet follows:

| Field | Length |
|---|---|
| Row Position | WORD |
| Column Position | WORD |

## Where

***Row Position***

is the row coordinate pointer screen position.

***Column Position***

is the column coordinate pointer screen position.

## Returns

This function returns to the caller the Current Pointer Screen Position.

## Remarks

The coordinate values are display mode dependent. Pixel values are returned if the display is in graphics mode. Character position values are returned if the display is in text mode.

# Category 7 – Function 68H

Gets the Current Pointer Shape

## Parameter Packet Format

The Parameter Packet is a location in application storage containing the pointer definition record. The pointer definition record is data describing a user buffer where the mouse device driver returns the Current Pointer Shape. This data has the following format:

| Field | Length |
|---|---|
| Buffer Length | WORD |
| Columns | WORD |
| Rows | WORD |
| Column Hot Spot | WORD |
| Row Hot Spot | WORD |

## Data Packet Format

The Data Packet is a location in application storage where the mouse device driver will return the Current Pointer Shape: a user buffer described by the data in the Parameter Packet.

## Where

***Buffer Length***

is the length of the (user) pointer shape buffer in BYTES.

When calling this function, the caller must specify the length of the pointer shape buffer in this field.

***Columns***

is the width in columns of the pointer image.

***Rows***

is the height in rows of the pointer image.

***Column Hot Spot***

is the column offset within the pointer image to the hotspot.

***Row Hot Spot***

is the row offset within the pointer image to the hotspot.

## Returns

If the Buffer Length value specified by the caller is smaller than the required storage for the pointer shape, the mouse device driver will return an error. The mouse device driver will return the minimum storage requirements for the pointer shape in the Buffer Length field.

If the Buffer Length value specified by the caller is greater than or equal to the amount of storage required for the pointer shape, the mouse device driver returns the pointer shape in the user's pointer shape buffer (that is, in the Data Packet) and the data describing the pointer shape in the pointer definition record (that is, in the Parameter Packet). The actual returned length is returned in the Buffer Length field.

## Remarks

The pointer shape buffer is described by the pointer definition record and for normal conditions consists of the Screen AND and Pointer XOR masks.

Return the mouse device driver level/version number.

## Parameter Packet Format

The Parameter Packet is a location in application storage with the following format:

| Field | Length |
|---|---|
| Dummy Address | DWORD |

## Data Packet Format

The Data Packet is a WORD in application storage where the mouse device driver returns its version number. A returned value of 1 indicates that the mouse supports all previous levels of support, as well as new OS/2 Version 1.2 Category 7 function, mouse device independence (see Mouse IDC Interfaces), and Display Independence (see VIO API for details).

OS/2 Version 1.0 and Version 1.1 Mouse Device Drivers will return an UNKNOWN COMMAND error to the caller for this function.

## Where

***Dummy Address***
consists of two WORDs of zeroes.

## Returns

None.

## Remarks

None.

# Category 8 Logical Disk Control IOCtl Commands

The following is a summary of Category 8 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 00H | Lock drive |
| 01H | Unlock drive |
| 02H | Redetermine media (End Format) |
| 04H | Begin Format |
| 03H | Set logical map |
| 20H | Block removable |
| 21H | Get logical map |
| 22H | Reserved |
| 43H | Set device parameters |
| 44H | Write track |
| 45H | Format and verify track |
| 5FH | Reserved |
| 63H | Get device parameters |
| 64H | Read track |
| 65H | Verify track |

Lock Drive

The operation of locking a drive is used to exclude I/O by another process on the volume in the drive. The Lock Drive call will succeed only if there is one file handle open on the volume in the drive; that is, the file handle on which this DosDevIOCtl call is issued. This is necessary since the desired result is to exclude all other I/O to this volume until the Unlock Drive call is issued.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***
is reserved and must be set to 0.

## Returns

See DosDevIOCtl error return codes.

## Remarks

None

# Category 8 – Function 01H

Unlock Drive

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***

is reserved and must be set to 0.

## Returns

See DosDevIOCtl error return codes.

## Remarks

This function releases a lock on a volume in a driver, thereby allowing I/O from other processes to that volume again.

The locked volume represented by the file handle on which this DosDevIOCtl call is issued must be in the drive.

Redetermine Media (End Format)

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***
is reserved and must be set to 0.

## Returns

See DosDevIOCtl error return codes.

## Remarks

This function rebuilds the device parameters (including the Volume Parameter Block (VPB) used by OS/2 to identify the volume), simulates a close on the current device handle, and forces a mount on the volume.

A Redetermine Media will dismount the volume from the current FSD, attach the new FSD, and reattach the current handle to the volume's new FSD.

The caller must have the disk or diskette locked when calling this function. Otherwise, the call will fail with the error ERROR_LOCK_VIOLATION.

The caller can have only one file open to refer to the disk or diskette. If other processes have the volume open, or the calling process has opened the volume multiple times, the call will fail returning ERROR_DRIVE_BUSY.

Redetermine Media is typically called after a new boot sector has been written to a volume after a format has been done.

# Category 8 –
# Function 03H

Sets the next logical drive letter that will be used to reference the drive.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Logical Drive Number | BYTE |

## Where

***Command Information***
is reserved and must be set to 0.

***Logical Drive Number***
On entry logical drive number (1 = A, 2 = B, etc) is specified.

On return, this byte specifies the logical drive currently mapped to the drive that the specified file handle is opened on. A zero is returned if there is only one logical drive mapped onto this physical drive.

## Returns

See DosDevIOCtl error return codes.

## Remarks

When copying diskettes on a drive whose physical driver number has more than one logical drive letter assigned to it, OS/2 issues diskette swap prompts to tell the user which logical driver letter is currently referencing the physical driver number. Set Logical Map is typically used to avoid this prompt by setting a driver number (corresponding to the driver letter) that will be referenced in the next I/O request.

The last logical driver letter assigned to the physical driver can be determined by calling the DosDevIoctl - Get Logical Map.

Begin format.

## Parameter Packet Format

| Field | Length |
|---|---|
| FSD name | ASCIIZ string |

## Data Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Where

***FSD name***
is the file system driver name, that is, the name the FSD exports.

A zero length string is used to indicate the FAT file system.

***Command Information***
is reserved and must be set to 0.

## Returns

See DosDevIOCtl error return codes.

## Remarks

This function attaches (mounts) a specified File System Driver (FSD) to a logical disk volume.

Begin Format is typically used to force mount the FSD a volume is being formatted for. It is called before starting a format operation on a volume.

Begin Format will unmount a current FSD (if any), and force a mount to the specified FSD.

A flag is set in the OS/2 kernel to indicate that a Begin Format (unmount/mount) was done. DosDevIoctl - Redetermine Media clears this flag, or performs an End Format. Therefore, a Redetermine Media MUST be done before the disk is DosClosed. Then the volume is ready for normal I/O service from the newly mounted FSD.

# Category 8 – Function 20H

This function is used to determine if a media is removable or fixed.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Data | BYTE |

## Where

***Command Information***

is reserved and must be set to 0.

***Data***

on return, the data byte is set accordingly:

0 = removable media
1 = nonremovable media

## Returns

See DosDevIOCtl error return codes.

## Remarks

None

Gets the logical drive letter that was last used to reference (open) the drive.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Logical Drive Number | BYTE |

## Where

***Command Information***

is reserved and must be set to 0.

***Logical Drive Number***

On entry, Logical Drive Number (1 = A, and others).

**Note:** This function will work as long as the file handle is valid. The file handle can be set to anything other than zero.

On return, if the device has more than one logical drive letter assigned to it, a drive number corresponding to the last drive letter that was used to reference (open) the device is returned. For example, if the entry drive number was 1(= A) and the returned was 2(= B), this drive was last referenced as the B: drive.

If only one drive letter (logical drive) is assigned to the device, a zero is returned.

## Returns

See DosDevIOCtl error return codes.

## Remarks

None

# Category 8 – Function 43H

Set the parameters for a specified device.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Extended BPB for devices | 31 BYTES |
| Number of cylinders | WORD |
| Device type | BYTE |
| Device attributes | WORD |

## Where

***Command Information***

the two low bits of the command byte are used to indicate one of three possible actions:

| *Bit Values* | *Description* |
|---|---|
| 00 | Revert to building the BPB off the medium for all subsequent Build BPB calls. This is used after a format operation to reset the device parameters to their original state. |
| 01 | Change the default BPB for the physical device. Changes the physical parameters for the drive - as opposed to parameters for the media in the drive. |
| 10 | Change the BPB for the medium to the specified BPB and return the new BPB as the BPB for the medium for all subsequent Build BPB calls. This is used to prepare the device for a format media operation according to the device parms specified. |

All other bits are reserved and must be set to 0.

***Extended BPB***

The term extended BPB refers to the BIOS parameter Block (table which describes the structure of the media), with some additional information describing the physical layout of the drive (heads, tracks, sectors).

The extended BPB has the following format:

| Field | Length |
|---|---|
| Bytes Per Sector | WORD |
| Sectors Per Cluster | BYTE |
| Reserved Sectors | WORD |
| Number of FATs | BYTE |
| Root Dir Entries | WORD |
| Total Sectors | WORD |
| Media Descriptor | BYTE |
| Sectors Per FAT | WORD |
| Sectors Per Track | WORD |
| Number Of Heads | WORD |

| Field | Length |
|---|---|
| Hidden Sectors | DWORD |
| Large Total Sectors | DWORD |
| Reserved | 6 BYTES |

***Number of cylinders***
indicates the number of cylinders defined for the physical device.

***Device type***
field describes the physical layout of the device specified. It takes one of the following values:

| *Value* | *Meaning* |
|---|---|
| 0 | 48 TPI low density diskette drive |
| 1 | 96 TPI high density diskette drive |
| 2 | Small (3 1/2 inch) (720KB) drive |
| 3 | 8 Inch Single Density floppy drive |
| 4 | 8 Inch Double Density floppy drive |
| 5 | Fixed disk |
| 6 | Tape drive |
| 7 | Other (other type of device) |

***Device attributes***
The device attributes is a bit field that returns various flag information about the specified drive:

| *Bit* | *Description* |
|---|---|
| 0 | Removable media flag. If set, the media can be changed |
| 1 | Changeline flag. If set, the media cannot be removed. |

All other bits are reserved and must be set to 0.

## Returns
See DosDevIOCtl error return codes.

## Remarks
None

Write Track, Read Track, Verify Track.

## Parameter Packet Format

These commands have the same parameter packet.

| Field | Length |
|---|---|
| Command Information | BYTE |
| Head | WORD |
| Cylinder | WORD |
| First Sector | WORD |
| Number of sectors | WORD |
| Track Layout Table | BYTES |

## Data Packet Format

The data packet is a buffer. For the write call it contains the data to be written. For a read call the buffer must be large enough to hold requested data. For the verify call the data packet is not used.

| Field | Length |
|---|---|
| Buffer | BYTES |

## Where

***Command Information***

is a byte with bit 0 defined as follows:

| *Bit* | *Description* |
|---|---|
| 0 | Clear - Track layout contains non-consecutive sectors or does not start with sector 1 |
| | Set - Track layout starts with sector 1 and contains only consecutive sectors |

All other bits are reserved and must be set to 0.

***Head***

is the physical head on the drive to perform the operation.

***Cylinder***

is the cylinder for the read/write/verify.

***First Sector***

is the logical sector number within the track layout table to start the I/O operation.

Note that the sector numbers are based from 0. (For example, the third sector is number 2.)

***Number of Sectors***

is the number of sectors to read/write/verify (up to the maximum specified in the track table - the IOCtl subfunctions will not step heads/tracks).

***Track Layout Table***

is as follows:

| Field | Length |
|---|---|
| Sector number for sector 1 | WORD |
| Sector size for sector 1 | WORD |
| Sector number for sector 2 | WORD |

| Field | Length |
|---|---|
| Sector size for sector 2 | WORD |
| Sector number for sector 3 | WORD |
| Sector size for sector 3 | WORD |
| ... | ... |
| Sector number for sector *n* | WORD |
| Sector size for sector *n* | WORD |

The sector table that is specified is used to provide information that will be used during the READ/WRITE/VERIFY track operations.

## Returns

See DosDevIOCtl error return codes.

## Remarks

This function is used to perform a write, read, or verify of sectors on a media.

This call performs the operation on the device that is specified in this request. The track table passed in on the call is used to determine the sector number which is passed to the disk controller for the operation. In cases where the sectors are oddly numbered or are non-consecutive, we break this request into N single sector operations and read/write/verify one sector at a time. Note also that the device driver will NOT correctly read a non-512 byte sector if the read operation would generate a DMA violation error. Application writers must be careful to make sure that this error does NOT occur. These DosDevIoctls are typically used when performing I/O outside of the normal file system data area on the media.

# Category 8 – Function 45H

Format and Verify a Track

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |
| Head | WORD |
| Cylinder | WORD |
| Number of Tracks | WORD |
| Number of sectors | WORD |
| Format Track Table | BYTES |

## Data Packet Format

| Field | Length |
|---|---|
| Starting Sector | BYTE |

## Where

The following fields are defined:

***Command Information***
is a byte with bit 0 defined as follows:

| *Bit* | *Description* |
|---|---|
| 0 | Clear - Track layout contains non-consecutive sectors or does not start with sector 1 |
| | Set - Track layout starts with sector 1 and contains only consecutive sectors |

All other bits are reserved and must be set to 0.

***Head***
is the physical head on the drive to perform the operation.

***Cylinder***
is the cylinder for the operation.

Upon return from a multi-track format request, if a defective spot is encountered on the media, the returned head and cylinder contain the defective area.

***Number of Tracks***
is the number of tracks to format/verify on a multi-track request. This is set to zero for single track requests.

On return from a multi-track request, Number of Tracks is set to one of the following values:

| Value | Description |
|---|---|
| 8000H | Multi-Track Successful |
| 4000H | Multi-Track Not Supported |
| 2000H | Multi-Track Failed on Format operation |
| 1000H | Multi-Track Failed on Verify operation |

***Number of Sectors***
is the number of sectors on the track being formatted.

On return from a multi-track request, if a defective spot is found on the media before the multi-track format operation can complete, this field indicates the number of tracks remaining (to be formatted) on this multi- track format request.

***Format Track Table***

is the format track table contains four byte tuples. Each tuple is in the form (c,h,r,n) with c = cylinder number, h = head number, r = sector id, and n = bytes per sector.

| *n* | *bytes / sector* |
|---|---|
| 0 | 128 |
| 1 | 256 |
| 2 | 512 |
| 3 | 1024 |

There is a 4-tuple for each sector in the track to be formatted. Both the head and cylinder numbers must be consistent within the tuple and with the corresponding parameter packet field.

***Starting Sector***

On input, this is the starting sector on a multi-track request. This is set to zero for a single track request.

On return from a multi-track format request, if a defective spot is found on the media, Starting Sector is the first logical sector number within the head and cylinder of the defective area.

## Returns

See DosDevIOCtl error return codes.

## Remarks

This routine formats and verifies the track specified according to the information passed in the Track Layout field. The track layout is passed to the controller and the controller performs the formatting. Note that some controllers do NOT support formatting tracks with varying sector sizes, so in general the application writer must take care to be sure that the sector sizes specified in the Track Layout table are all the same.

# Category 8 – Function 63H

Get Device Parameters

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Extended BPB for device | 31 BYTES |
| Number of cylinders | WORD |
| Device type | BYTE |
| Device attributes | WORD |

## Where

The following fields are defined:

***Command Information***

is a BYTE with bit 0 defined as follows:

| *Value* | *Description* |
|---|---|
| 0 | Return the recommended BPB for the drive. The recommended BPB for the drive is the BPB for the physical device, unless it is a formatted fixed media. Then it is the BPB that was on the media when the system was booted. |
| 1 | Return the BPB for the media currently in the drive. This always reads the BPB off the current media in the drive. An error is returned if the media is unformatted. |

All other bits are reserved and must be set to zero.

***Extended BPB for device***

The device driver maintains two BPB's for each drive. One is the current BPB (that corresponds to the media in the drive). The other is a recommended BPB that is based on the type of media that corresponds to the physical device (for a high density drive, that is a BPB for a 96 tracks-per-inch (TPI) floppy, for a low density drive it is the BPB for a 48 TPI floppy, etc). The low bit of the command information field indicates which BPB the application would like to see.

***Number of cylinders***

The number of cylinders indicates the number of cylinders defined for the physical device.

***Device type***

The device type field describes the physical layout of the device specified. It takes one of the following values:

| *Value* | *Meaning* |
|---|---|
| 0 | 48 TPI low density diskette drive |
| 1 | 96 TPI high density diskette drive |
| 2 | Small (3 1/2 inch) (720KB) diskette drive |
| 3 | 8 Inch Single Density diskette drive |
| 4 | 8 Inch Double Density diskette drive |
| 5 | Fixed disk |
| 6 | Tape drive |
| 7 | Other (other type of device) |

***Device attributes***

The device attributes is a bit field that returns various flag information about the specified drive:

| *Bit* | *Description* |
|---|---|
| 0 | Removable media flag If set, the media can be changed |
| 1 | Changeline flag If set, the media cannot be removed |

All other bits are reserved and must be set to 0.

## Returns

See DosDevIOCtl error return codes.

## Remarks

This function gets the parameters for a specified device.

# Category 9 Physical Disk Control IOCtl Commands

Category 9 is a category which is used to access physical partitionable fixed disks.

The handle, used for Category 9 commands is the handle returned by the DosPhysicalDisk (function 2) API function call. This handle is used to tell the system which physical disk the IOCtl command is for.

The physical disk control commands relate to the entire partitionable fixed disk. Direct track and sector I/O begin at the beginning of the physical drive. Function 63H, get physical device parameters, describes the entire physical device.

The following is a summary of Category 9 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 00H | Lock physical drive |
| 01H | Unlock physical drive |
| 44H | Physical write track |
| 63H | Get physical device parameters |
| 64H | Physical read track |
| 65H | Physical verify track |

Lock Physical Drive

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***
is reserved and must be set to 0.

## Returns

See DosDevIOCtl error return codes.

## Remarks

All the logical units on the physical drive are affected as well.

# Category 9 – Function 01H

Unlock Physical Drive

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***
is reserved and must be set to 0.

## Returns

See DosDevIOCtl error return codes.

## Remarks

All the logical units on the physical drive are affected as well.

Physical Write Track, Physical Read Track, Physical Verify Track

## Parameter Packet Format

These commands have the same parameter packet.

| Field | Length |
|---|---|
| Command Information | BYTE |
| Head | WORD |
| Cylinder | WORD |
| First Sector | WORD |
| Number of sectors | WORD |
| Track Layout Table | BYTES |

## Data Packet Format

The data packet is a buffer. For the WRITE call it contains the data to be written. For a READ call the buffer must be large enough to hold requested data. For the VERIFY call the data packet is not used.

| Field | Length |
|---|---|
| Buffer | BYTES |

## Where

***Command Information***

is a bit field as follows:

| *Bit* | *Description* |
|---|---|
| 0 | 0 (Clear) - Track layout contains non-consecutive sectors or does not start with sector 1 |
| | 1 (Set) - Track layout start with sector 1 and contains only consecutive sectors |
| All other bits | are reserved and must be 0. |

***Head***

is the physical head on the drive to perform the operation.

***Cylinder***

is the cylinder for the read/write/verify.

***First Sector***

is the logical sector number within the track layout table to start the I/O operation.

Note that the sector numbers are based from 0. (For example, the third sector is numbered 2.)

***Number of Sectors***

The number of sectors to read/write/verify (up to the maximum specified in the track table - the IOCtl subfunctions will NOT step heads/tracks).

***Track Layout Table***

The track layout table is as follows:

| Field | Length |
|---|---|
| Sector number for sector 1 | WORD |
| Sector size for sector 1 | WORD |
| Sector number for sector 2 | WORD |

## Category 9 –
## Functions 44H, 64H, 65H

| Field | Length |
|---|---|
| Sector size for sector 2 | WORD |
| Sector number for sector 3 | WORD |
| Sector size for sector 3 | WORD |
| ... | ... |
| Sector number for sector N | WORD |
| Sector size for sector N | WORD |

### Returns

See DosDevIOCtl error return codes.

### Remarks

This call will perform the operation on the physical drive that is specified in this request. It works like the similar Category 8 command except that the I/O is done offset from the beginning of the physical drive instead of from the beginning of the extended volume associated with the unit number (category 8).

The track table passed in on the call is used to determine the sector number which is passed on to the disk controller for the operation. In cases where the sectors are oddly numbered or are non-consecutive the request is broken into N single sector operations, and read/written/verified one sector at a time. Note also that the device driver will not correctly read a non 512 byte sector if the read operation would generate a DMA violation error. Application writers must be careful to make sure that this error will not occur.

The sector table that is specified is used to provide information that is used during the READ/WRITE/VERIFY track operations.

Get Physical Device Parameters

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Reserved | WORD |
| Number of Cylinders | WORD |
| Number of Heads | WORD |
| Number of Sectors per Track | WORD |
| Reserved | WORD |
| Reserved | WORD |
| Reserved | WORD |
| Reserved | WORD |

## Where

***Command Information***
is reserved and must be set to 0.

***Number of Cylinders***
is where the number of cylinders on the physical drive are returned.

***Number of Heads***
is where the number of heads on the physical drive are returned.

***Number of Sectors per Track***
is where the number of sectors per track on the physical drive are returned.

## Returns

See DosDevIOCtl error return codes.

## Remarks

The data values returned apply to the entire physical disk.

# Category 10 Character Device Monitor IOCtl Commands

The following is a summary of Category 10 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 40H | Register a Monitor |

Register a Monitor

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

The Data Packet includes the following parameters passed from a DosMonReg function call:

| Field | Length |
|---|---|
| Placement flag | WORD |
| Index | WORD |
| Address of input buffer | DWORD |
| Offset of output buffer | WORD |

## Where

***Command Information***
is reserved and must be set to 0.

***Placement flag***
is the DosMonReg function call parameter that is used by an application to indicate:

- Where its monitor buffers are to be placed within a monitor chain, relative to monitors already registered on the monitor chain.
- What special processing requirements need to be supported by the monitor dispatcher.

Refer to the DosMonReg function call description in the *OS/2 Version 1.2 Programming Reference* for valid parameter values.

***Index***
is the DosMonReg function call parameter that is used by an application to indicate on which monitor chain its monitor buffers are being registered. The accepted values for this parameter vary for each device driver. See "Mouse Device Driver," "Console Device Drivers (Screen and Keyboard)," and "The Printer Device Driver" in this book to determine the valid parameter value for each device driver.

***Address of input buffer***
is the DosMonReg function call parameter that specifies the address of a monitor input buffer allocated by an application, initialized by the monitor dispatcher, and used by DosMonRead function calls.

***Offset of output buffer***
is the DosMonReg function call parameter that specifies the offset of a monitor output buffer allocated by an application in the same data segment as the input buffer, initialized by the monitor dispatcher, and used by DosMonWrite function calls.

**Note:** Refer to the Chapter 8, "Character Device Monitors" in this book and the descriptions of the DosMonReg, DosMonRead, and DosMonWrite function calls in the *OS/2 Version 1.2 Programming Reference* for more information.

# Category 10 – Function 40H

## Returns

An error is returned to the caller if monitor registration fails because of:

- Invalid parameters; for example, specification of an invalid monitor chain (INDEX), monitor buffers too small.
- Not enough resources; monitor registration requires additional system resources.

## Remarks

Character device drivers that support device monitors receive this request when an application calls DosMonReg to register a monitor. The device driver places the values of these parameters in registers and calls the Register device helper routine to complete monitor registration.

See Chapter 8, "Character Device Monitors" for detailed information on character device monitors and monitor support in character device drivers.

# Category 11 General Device Control IOCtl Commands

The following is a summary of Category 11 IOCtl commands:

| *Function* | *Description* |
|---|---|
| 01H | Flush input buffer |
| 02H | Flush output buffer |
| 41H | Session switch or termination notification |
| 60H | Query monitor support |

# Category 11 – Function 01H

Flush Input Buffer.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***

is reserved and must be set to 0.

## Returns

None

## Remarks

None

Flush output buffer.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***
is reserved and must be set to 0.

## Returns

None

## Remarks

None

# Category 11 – Function 41H

Session switch or termination notification.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | 4 WORDS |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***

contains a data structure defined as follows:

WORD - structure length (8 bytes)
WORD - notification action type
1 = pre session save notification
2 = post session save notification
4 = post session restore notification
8 = session termination notification
WORD - incoming session ID
WORD - outgoing session ID (type 2)

## Returns

None

## Remarks

Screen switching notification will not be given when switch between non-full screen sessions (that is, Presentation Manager and text-windowed sessions). Termination notification will be given in all cases of session termination.

Query monitor support.

## Parameter Packet Format

| Field | Length |
|---|---|
| Command Information | BYTE |

## Data Packet Format

| Field | Length |
|---|---|
| Set To 0 | BYTE |

## Where

***Command Information***
is reserved and must be set to 0.

## Returns

None

## Remarks

This request is used to query a device driver for monitor support.

The device driver should return the system error, Monitors-Not-Supported, if it does not support character monitors. If monitors are supported, then it should return No-Error (00H).

# Chapter 8. Character Device Monitors

OS/2 provides a mechanism for applications to directly access and intercept data flowing through data streams belonging to some character device drivers. (See Figure 8-1.) This mechanism allows applications to filter (remove, insert, or modify) data passing through a character device by registering one or more character device monitors with the device driver. The mechanism requires support by the character device driver, as well as by the application.

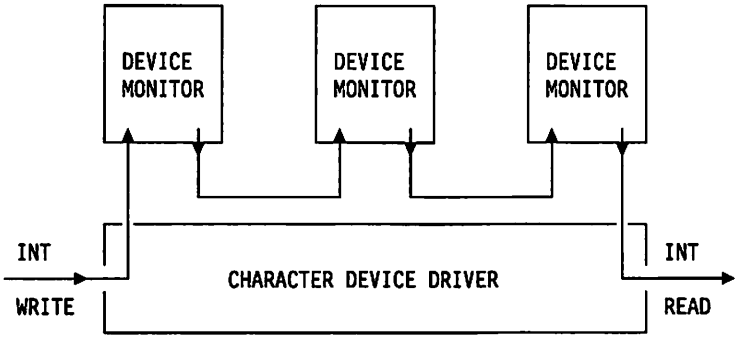



Figure 8-1. Data interception by device monitors

Applications that monitor data passing through a character device can perform a variety of functions. They can serve as:

- User-extensions of a character device driver.

  Data passing through a data stream may be manipulated to modify the state of the device as viewed by another application.

- Dialog managers.

  Command sequences or control keys can be activated on a given keystroke. These include pop-up facilities and note-pad applications.

- Language translators.

  Characters can be translated from one language to another.

- Redirection mechanisms.

  Data can be redirected from one device to another by character device monitors, as illustrated in Figure 8-2 on page 8-2. An application may register two monitors with two separate character devices. Data taken from the data stream belonging to the first device by the first monitor may be placed into the data stream belonging to the second device by the second monitor.

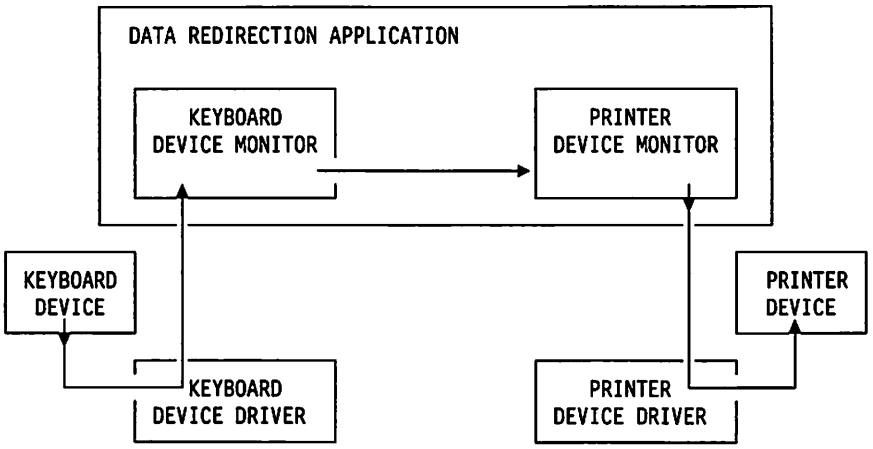



Figure 8-2. Data redirection using device monitors

OS/2 character device drivers that provide monitor support include: the keyboard, mouse and printer device drivers. OS/2 character device monitors that are part of the OS/2 product include:

*OS/2 Print Spooler*, that registers two monitors for each PRINTER device. The monitor takes data from the PRINTER device data stream and spools or stores the data in files on a fixed disk. The monitor sends SPOOLER data files to the PRINTER device for printing by returning the data to the data stream.

This chapter describes the OS/2 monitor mechanism, which includes the character device monitor process and the support required by a character device driver that enables device monitoring to be performed. Guidelines are given for developing a character device monitor and for implementing monitor support in the character device driver.

# Monitoring Character Device Data Streams

A device driver receives data from a device or receives requests from applications to write data to a device. The keyboard and mouse device drivers receive data from the keyboard and mouse devices, respectively. The printer device driver receives requests from applications to write data to the printer device.

The flow of data between a device driver and its device is called a data stream. The device driver defines the data streams for its device and associates the data it receives from a device or an application with a particular data stream.

There are several ways in which a device driver may define its data streams. When the keyboard or mouse device driver receives data from its device, it associates that data with the current foreground OS/2 session (which may be the Presentation Manager session) and places the data into the data stream defined for that session. These device drivers define data streams for each session. When the printer device driver receives a request from an application to write data to the printer, it places the data into the data stream defined for that printer device. This device driver defines data streams for each physical device.

A single data stream may be filtered by one or more character device monitors that are linked together in a monitor chain. That is, the first character device monitor in a monitor chain receives data directly from the device driver. This character device monitor filters the data and passes it on to the next character device monitor in the chain. This character device monitor filters the filtered data and passes it on to the next character device monitor in the chain; and so forth. The last character device monitor in the chain passes the filtered data back to the device driver.

Just as device drivers define their data streams, device drivers determine how character device monitors are chained together. The keyboard and mouse device drivers, for example, support a monitor chain for each OS/2 session, including the Presentation Manager session. The printer device driver supports two monitor chains for each physical printer device. See "Mouse Device

Driver," "Console Device Drivers (Screen and Keyboard)," and "The Printer Device Driver" in this book for a description of how these device drivers support device monitors.

## Device Monitor Support Limitations

Data streams for character devices may be monitored by a character device monitor, only when the character device driver provides device monitor support. Applications, including both OS/2 and Presentation Manager applications, may monitor keystrokes and mouse clicks for all OS/2 sessions, including the OS/2 session in which all Presentation Manager applications run. However, applications, including both OS/2 and Presentation Manager applications, may not monitor keystrokes or mouse clicks for Presentation Manager sessions.

The keystroke and mouse click mechanism for Presentation Manager sessions differs from the mechanism for OS/2 sessions, as illustrated in Figure 8-3 on page 8-4. The keyboard and mouse device drivers manage keystrokes and mouse clicks, respectively, for each OS/2 session. These device drivers create monitor chains for each OS/2 session, and applications may register keystroke and mouse device monitors for each OS/2 session.

The Presentation Manager manages both keystrokes and mouse clicks for each of its extended sessions. The Presentation Manager does not create monitor chains for each of its extended sessions. Applications may not register a keystroke or mouse device monitor for a Presentation Manager extended session.

All applications (including OS/2 and Presentation Manager applications) may, however, monitor printer data. The printer device driver bases the definition of its data streams on the physical device. In general, data streams for character devices that do NOT base the definition of their data streams on sessions may be monitored by character device monitor, when the character device driver provides device monitor support.

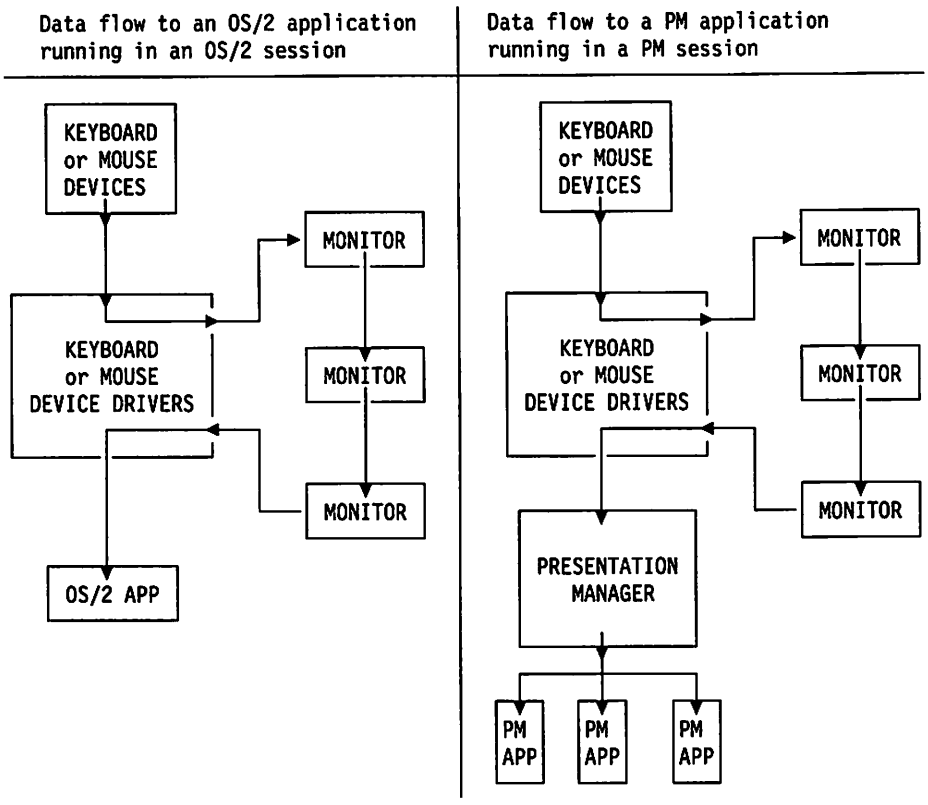



Figure 8-3. OS/2 Monitors and PM Applications

**Note:** A Presentation Manager application may filter keyboard and mouse events (messages) through hook procedures. These procedures are registered by an application and are called when certain events occur. More than one procedure may be called when a single event occurs. In this case, procedures are chained together so that each event is passed first to one procedure and then to the next, and so on down the chain. This chain of procedures is called a *hook chain*.

There are two kinds of hook procedures: system hook procedures and queue hook procedures. System hook procedures are called when an event occurs in the Presentation Manager's system queue. Queue hook procedures are called when an event occurs in the Presentation Manager application's queue.

There are several kinds of events that can be hooked. These include messages input to an application and messages sent by an application.

For a detailed description of hook procedures refer to the *OS/2 Version 1.2 Programming Guide*.

# The OS/2 Monitor Mechanism

The OS/2 monitor mechanism consists of:

- Monitor buffers
- Monitor threads
- Signals.

This mechanism is supported by the OS/2 Monitor Dispatcher, which consists of:

1. *The OS/2 monitor dispatcher function calls*, a set of five dynamic link routines:

   DosMonOpen
   DosMonReg
   DosMonClose
   DosMonRead
   DosMonWrite.

2. *The Monitor Dispatcher Device Helper*, a set of five OS/2 device helper routines:

   MonitorCreate
   Register
   DeRegister
   MonWrite
   MonFlush.

The OS/2 monitor dispatcher function calls are part of the MONCALLS dynamic link function call library. These routines are loaded on demand at privilege level 2 and are callable from applications loaded at privilege level 2 or 3. Using these routines, applications may intercept and filter data passing through a device. The OS/2 monitor function calls provide the interface for monitor applications to interact with:

- The device driver, to request registration and termination of monitor activity
- The data stream, directly through its own monitor input and output buffers.

The monitor dispatcher device helper routines are part of the OS/2 device helper, available to all character device drivers. Through these routines, a character device driver provides the support for applications that wish to monitor its data streams. The monitor dispatcher device helper provides the interface for character device drivers to interact with:

- The monitor dispatcher on behalf of itself and applications requesting registration and termination of monitors
- Applications monitoring its data streams.

In addition, the monitor dispatcher device helper provides the mechanism for passing data from one monitor to another.

Character device monitor applications register a pair of *monitor buffers* (an input and an output buffer) with a specific data stream for a specific character device. These buffers become part of a chain of buffers, that is, a *monitor chain*. The OS/2 monitor dispatcher manages these buffers on behalf of the character device monitor applications, and manages the monitor chains on behalf of the character device driver.

The character device driver places data for a specific data stream into the monitor chain associated with the data stream. Each monitor in the monitor chain receives data from the data stream through its registered monitor input buffer. Each monitor in the monitor chain returns filtered data to the data stream through its registered monitor output buffer.

Character device monitor applications are generally multi-threaded, with child *monitor threads* that take data from the data stream (that is, from a monitor input bufffer) and return filtered data to the data stream (that is, to a monitor output buffer).

Monitor threads utilize *signals* in two ways:

- When no data is available for the monitor in the data stream, a monitor thread may wait for a signal from the monitor dispatcher when data is available for the monitor. When the data stream cannot accept data from the monitor (for example, there is a blockage downstream in the monitor chain or at the device driver), a monitor thread will wait for a signal from the monitor dispatcher when the data stream can accept data from the monitor.
- When a monitor thread has finished taking data from or returning data to the data stream it signals the monitor dispatcher that it has completed its work.

Signaling between monitor threads is managed by the monitor dispatcher and is transparent to the monitor application.

## The Character Device Monitor Process

A character device monitor is an application, or part of an application, that uses standard OS/2 function calls to interact with the device driver and its data streams. A character device monitor may monitor only one data stream belonging to a character device driver. An application, however, may include one or more character device monitors, monitoring one or more data streams belonging to one or more character device drivers. For example, an application may monitor keystrokes and mouse clicks for each OS/2 session. This application will include a character device monitor for each OS/2 session for each of the keyboard and mouse device drivers.

To monitor a data stream belonging to a character device driver, an application must first gain access to that data stream.

*An application gains access to a data stream* by calling:

1. *DosMonOpen* to establish a connection to the device driver.

   DosMonOpen returns a device handle for monitors to the application. The application will use this handle to direct subsequent DosMonReg and DosMonClose calls to the device driver.

2. *DosMonReg* to register a pair of monitor input and output buffers with a particular data stream belonging to the device driver.

   Once the monitor is installed in the monitor chain, the monitor dispatcher automatically moves data between monitors (if there are more than one in the chain) and returns filtered data to the device driver.

After an application has gained access to a data stream, it may remove, insert, modify, or view all characters passing through the data stream.

*An application intercepts data flowing through a data stream by calling*:

1. *DosMonRead* to take data from the data stream (that is, from its own monitor input buffer) and place it into a private data area where the application can access it freely for filtering; and
2. *DosMonWrite* to take filtered data from the application's private data area and return it to the data stream (that is, into its own monitor output buffer).

When an application no longer wants to monitor data streams belonging to a single character device driver, it must relinquish access to all data streams belonging to that device driver, in addition to terminating its monitor threads.

*An application terminates a monitor* by calling:

1. *DosMonClose* to close the device handle (that is, terminate the connection to the device driver for the monitors); and
2. *DosExit* to terminate its monitor threads that are directly accessing the data streams.

Figure 8-4 on page 8-7 illustrates pseudocode for a simple character device monitor:

*A simple, single-threaded Character Device Monitor*

Get access to the device's data stream:

CALL DosMonOpen to open the device and get a handle for monitors.

CALL DosSetPrty to set the priority of our monitor thread high.

CALL DosMonReg to register a pair of input and output buffers as
a monitor for the device's data stream.

WHILE we want to monitor the data stream:

CALL DosMonRead to read a data record from the monitor's input buffer.

Filter the data record.

CALL DosMonWrite to return the filtered data record to the monitor's
output buffer.

END WHILE

We're done monitoring the data stream:

CALL DosMonClose to close the monitor handle to the device.

CALL DosExit to terminate this application.

Figure 8-4. Pseudocode for a simple character device monitor

## The Character Device Driver with Monitor Support

For an application to monitor data passing through a character device, the character device driver must provide monitor support.

For each data stream that can be monitored by applications, the character device driver must first create a *monitor chain*. For each monitor chain, the character device driver must define a *monitor chain buffer* in one of its DATA segments, where the monitor dispatcher will place filtered data that has passed through all monitors registered with the monitor chain. In addition, the character device driver must define a *notification routine* within one of its CODE segments. This routine is called by the monitor dispatcher when filtered data has been placed into the monitor chain buffer.

*A character device driver creates a monitor chain by calling the MonitorCreate device helper routine.*

When an application registers a monitor (that is, calls DosMonReg), the character device driver receives an IOCTL request from the monitor dispatcher on behalf of the application to register the monitor with the monitor chain belonging to one of its data streams.

*A character device driver registers a monitor with the monitor chain belonging to one of its data streams by calling the Register device helper routine.*

When one or more monitors are registered with a monitor chain belonging to one of its data streams, a character device driver can send data to its monitors.

*A character device driver sends data to monitors registered with one of its monitor chains by calling the MonWrite device helper routine.*

Under certain conditions a character device driver may guarantee that all data sent to its monitors in a monitor chain has been filtered and returned to the data stream.

*A character device driver flushes all data from all monitors registered with a monitor chain by calling the MonFlush device helper routine.*

When a monitor application stops monitoring data streams belonging to a character device driver OR when a monitor application abnormally terminates (for example, the user presses the CTRL-C key sequence), the character device driver receives a monitor close request from the file system. Because an application may have one or more monitors registered with one or more monitor chains belonging to the device driver, The character device driver must remove all monitors belonging to the application that are registered on any of its monitor chains.

*A character device driver removes monitors belonging to an application by calling the DeRegister device helper routine for each monitor chain belonging to each of its data streams.*

## The Relationship between a Character Device Driver and Its Monitors

A character device driver and its monitors are interdependent. The implementation of the character device driver determines the ground rules for its monitors, including:

- The definition of its data streams and monitor chains
- The format of the data flowing through its monitor chains
- The rules on consuming, modifying, and returning data.

Because a monitor directly interacts with a data stream, there is a danger of severely impacting the data stream if a monitor is not well-behaved, that is, if the monitor does not adhere to the character device driver's rules.

The interdependence of a character device driver and its monitors can be directly demonstrated in two ways: by presenting a description of the sequence of events occurring in the character device driver and its monitors during

- Monitor registration and termination
- Movement of data through a monitor chain.

### Registering and Terminating a Monitor

During monitor registration and termination, the monitor dispatcher communicates with the character device driver on behalf of the monitor application through the OS/2 file system and IOCTL interface. The application initiates OS/2 monitor function calls and the character device driver receives and responds to the corresponding requests. Figure 8-5 on page 8-9 illustrates the monitor function calls made by the application and the corresponding responses by the character device driver.

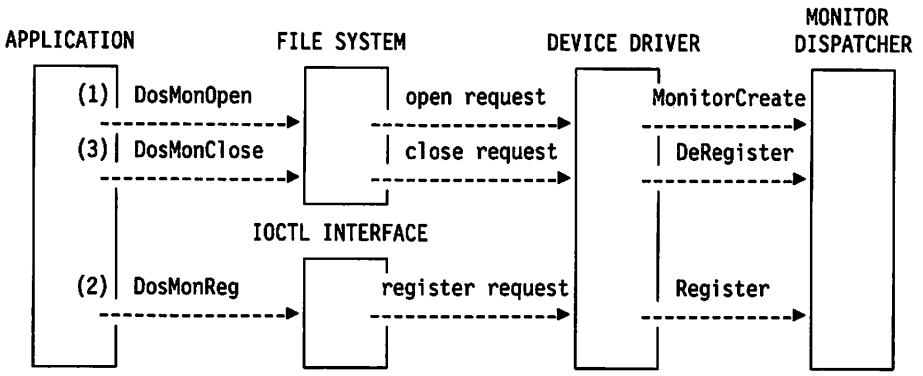



Figure 8-5. Application to device driver interface

Character device driver Z defines only one data stream for its device. Application X wishes to monitor character device driver Z's data stream.

1. Application X calls *DosMonOpen* to get a handle to character device driver Z.

   Through the file system, the monitor dispatcher sends an open request for monitors to character device driver Z, on behalf of application X.

   If character device driver Z has not already created a monitor chain for its data stream, character device driver Z may call the *MonitorCreate* device helper to create the monitor chain.

2. Application X calls *DosMonReg* to register Monitor Y with the monitor chain for character device driver Z's data stream.

   On behalf of application X the monitor dispatcher issues a monitor register IOCTL request to character device driver Z.

   If character device driver Z has not already defined a monitor chain for its data stream, character device driver Z must now call the *MonitorCreate* device helper to create the monitor chain.

   On receiving the monitor register IOCTL request, character device driver Z calls the *Register* device helper so that the monitor dispatcher can install the monitor buffers belonging to Application X in the monitor chain.

3. When Application X stops monitoring character device driver Z's data stream, Application X calls *DosMonClose*.

   Through the file system the monitor dispatcher sends a close request for monitors to character device driver Z on behalf of Application X.

   On receiving the monitor close request, character device driver Z must call the *DeRegister* device helper so that the monitor dispatcher can remove the monitor buffers belonging to Application X from the monitor chain.

## Data Passing through a Monitor Chain

A character device can be an *input device* or an *output device*. A device driver receives data from an input device (for example, the keyboard or mouse) and makes it available to users through its API buffers. A device driver receives requests from applications to send data to an output device (for example, the printer).

When character device driver Z has created a monitor chain for its data stream and application X has registered monitor Y with that monitor chain, data flows through the monitor chain as illustrated in Figure 8-6 on page 8-10.

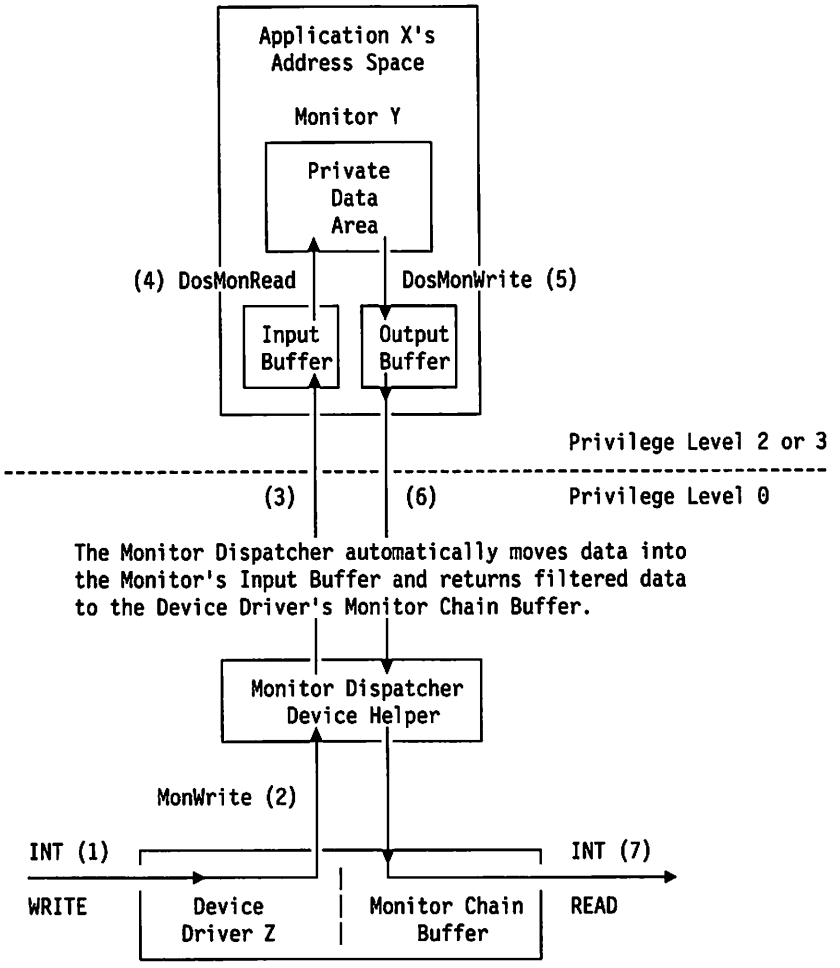



Figure 8-6. Data flow through a monitor chain

1. Character device driver Z receives data.

   If character device Z is an input device, the data received by its device driver comes directly from the device.

   If character device Z is an output device, the data received by its device driver comes from an application requesting output to the device.

2. Character device driver Z determines where to place the data, that is, into which data stream and monitor chain, if any, to place it. In this example, character device driver Z has defined only one data stream for its device.

   Because a monitor chain has been created for the data stream, character device driver Z places the data into the monitor chain by calling the *MonWrite* device helper.

3. Since monitor Y is registered with the monitor chain, the monitor dispatcher automatically moves the data into the input buffer belonging to Monitor Y.

4. Monitor Y calls *DosMonRead* to take data from the data stream (that is, from its own monitor input buffer) and place it into a private data area where it can freely access and process the data.

5. When Monitor Y has processed, or filtered, the data, Monitor Y returns the filtered data to the data stream (that is, to its own monitor output buffer) by calling *DosMonWrite*.

6. Since Monitor Y is the only monitor registered with the monitor chain, the monitor dispatcher automatically returns the filtered data from Monitor Y's output buffer to character device driver Z's *monitor chain buffer* and calls character device driver Z's *notification routine*.

7. Character device driver Z processes the data received in its monitor chain buffer before returning to the monitor dispatcher.

   If character device Z is an input device, character device driver Z moves the filtered data from its monitor chain buffer into its API buffer, where it can be READ from an application.

   If character device Z is an output device, character device driver Z sends the filtered data to the device.

# Developing a Character Device Monitor

## OS/2 Monitor Function Calls

An application process that monitors data passing through a character device uses the OS/2 monitor function calls to:

1. Gain access to the data stream (DosMonOpen, DosMonReg).
2. Take data from the data stream (DosMonRead) and return filtered data to the data stream (DosMonWrite).
3. Relinquish access to the data stream (DosMonClose).

Monitor applications use the OS/2 monitor function calls to communicate with a character device driver and to directly intercept data from a data stream belonging to the character device driver.

The OS/2 monitor function calls are part of the OS/2 monitor dispatcher. They are shipped with OS/2 as the MONCALLS dynamic-link subroutine library. These routines are loaded at privilege level 2 and are callable from applications running at privilege level 2 or 3.

Each OS/2 monitor function call utilizes the OS/2 System Trace Facility, tracing entry and exit parameters when tracing is enabled for monitors. This facility provides assistance for the monitor application developer during the debug phase of the application's development cycle.

During each monitor function call, the monitor dispatcher performs parameter validation which includes whether:

- A device handle has been previously opened
- Monitor buffers have been previously registered.

Refer to the individual monitor function call descriptions which follow for details on parameter validation.

OS/2 API standards require that a general protection (GP) fault occurs when a null selector is passed as a address parameter to an API call. To conform to this standard, the monitor function calls generate a GP fault when a null selector is passed as an address parameter. A pop-up message notifies the user that the process is being terminated.

The monitor dispatcher manages all monitor input and output buffers registered with a monitor chain. The monitor dispatcher moves data to and from each buffer, maintaining pointers to the data and data lengths for each data record. The content of each data record is irrelevant to the monitor dispatcher. The content of a data record is the responsibility of the character device driver and its monitors.

Although a monitor application should not touch its monitor input and output buffers after monitor registration, the monitor dispatcher cannot prevent this from happening. To prevent system failure from occurring, the monitor dispatcher checks the integrity of the data within these buffers with respect to the pointers and data lengths it maintains. If corruption of these pointers and data lengths is detected, the monitor dispatcher forces a GP fault to occur so that the process is terminated abnormally. A pop-up message notifies the user that the process is being terminated.

## DosMonOpen

An application that monitors data passing through a character device must first *get a handle to the device* by calling DosMonOpen. The application will need this handle to communicate with the character device driver in subsequent monitor function calls: DosMonReg (for monitor registration) and DosMonClose (for monitor termination).

On calling DosMonOpen, the application pushes the parameters onto the stack, as shown in Table 8-1. The address parameters are *selector:offset* addresses. Refer to the Programming Reference Library for a detailed description of the parameters to this call.

Table 8-1. DosMonOpen parameter stack

| Parameter description | Size |
|---|---|
| Address of device name string | DWORD |
| Address of device handle returned | DWORD |

DosMonOpen parameter validation includes verifying that:

1. A *device*, not a *file*, is being opened; and
2. The application does not currently have a handle open for this device.

Null selectors passed as addresses for these variables will cause a GP fault to occur. A pop-up message notifies the user that the process is being terminated.

DosMonOpen calls on the OS/2 file system to open a handle to a character device *for monitors* by calling DosOpen. DosMonOpen returns the device handle returned from DosOpen to the application. DosMonOpen calls DosOpen with the third bit of the DosOpen Open Mode parameter set, to indicate to the file system that a monitor open request must be sent to the device driver. (This differentiates this request from a normal open request.) On receiving the monitor open request, the character device driver may (optionally) issue a MonitorCreate device helper call to create a monitor chain. A character device driver may create a monitor chain *anytime prior to* issuing a Register device helper call, including at initialization time during device driver installation.

**Note:** An application needs to call DosMonOpen only once per character device. Repeated DosMonOpen calls by an application to open the same device for monitors will return the same handle.

An application may register one or more monitors with data streams belonging to the same character device using the same handle. For example, an application may open the keyboard device for monitors (call DosMonOpen). Using the handle returned from this call, the application may register a monitor for each OS/2 session. (Each OS/2 session has its own keystroke data stream and associated monitor chain.)

## DosMonReg

Before making this call, an application previously must have called DosMonOpen to get a handle to the character device driver. The application needs this handle so that a monitor register request can be sent to the character device driver.

After an application has called DosMonOpen, it may register a pair of input and output buffers as a monitor in a monitor chain associated with one of the character device driver's data streams. Data from the data stream will be redirected through all buffers in the monitor chain. Each monitor in the chain accesses data in the data stream by calling DosMonRead to take a data record from its input buffer. The monitor returns filtered data to the data stream by calling DosMonWrite to place the filtered data record into its output buffer.

On calling DosMonReg, the application pushes the parameters onto the stack, as shown in Table 8-2 on page 8-13. The address parameters are *selector:offset* addresses. Refer to the Programming Reference Library for a detailed description of the parameters to this call.

| Table 8-2. DosMonReg parameter stack | |
|---|---|
| **Parameter description** | **Size** |
| Device handle, returned from DosMonOpen | WORD |
| Address of monitor's input buffer | DWORD |
| Address of monitor's output buffer | DWORD |
| Positional placement flag | WORD |
| Index, defined by individual device driver | WORD |

DosMonReg parameter validation includes verifying that:

1. The handle is currently opened for the device.
2. The application did not previously register the buffers as monitors for any character device.
3. The buffers are in the same segment, do not overlap, and are as large as specified.

Null selectors passed as addresses for the monitor's input or output buffers will cause a GP fault to occur. A pop-up message notifies the user that the process is being terminated.

The monitor's input and output buffers are allocated by the application from the same data segment. The first WORD of each buffer must contain the length of the buffer (in bytes), length WORD inclusive. To maintain data movement through all monitor buffers in a monitor chain (more than one application may register monitors with the same monitor chain) the *minimum* size of these buffers must be the length of the character device driver's monitor chain buffer *plus 20 bytes*. The monitor chain buffer is located at the end of a monitor chain and receives filtered data that has passed through all monitors in the monitor chain. Because it is defined and owned by the character device driver, its length is documented in the descriptions of the individual character device drivers that support monitors. See the descriptions of the keyboard, mouse and printer device drivers in this book for this information.

**Note:** The monitor dispatcher manages these buffers on behalf of the monitor application. After monitor registration, the application should not touch these data areas. Corruption of the monitor buffers by the application will result in a GP fault, termination of the application, and notification (through a popup message on the screen) that the application is being terminated.

Because monitor buffers effectively become part of a data stream, a pair of monitor input and output buffers can be registered by a single application to monitor only one device data stream at a time. This ensures that data from two separate data streams does not become intermixed. However, because the application owns the monitor buffer segment, the monitor dispatcher cannot prevent an application from sharing it with another application. To prevent intermixing of data streams by separate applications sharing monitor buffers, an application should not share with another application the data segment in which its monitor buffers are allocated.

The *index* parameter indicates the data stream for the character device that the application is monitoring. *The individual character device driver defines this parameter.* For example, *index* indicates the OS/2 session number for the keyboard and mouse device drivers. See the descriptions of the keyboard, mouse and printer device drivers in this book for details.

The positional placement indicator is used to specify:

1. Placement of a monitor's buffers within the monitor chain (FIRST, LAST, or DEFAULT)
2. Special processing requirements.

Location of a monitor within a monitor chain is relative to monitors that are already registered with the monitor chain. See "Positioning of Monitors in a Monitor Chain" on page 8-19 in this chapter for details.

A monitor that may be the only monitor in a monitor chain will not receive data from the data stream (that is, from a DosMonRead call) when the character device driver cannot process the filtered data that it has received from the monitor chain. The design of the monitor, however, may require that it continue receiving data, even though a blockage has occurred at the device. The monitor dispatcher accommodates this requirement when a monitor registers with a special case positional placement indicator. See "Special Processing Requirements" on page 8-22 in this chapter for an example.

**Note:** Any monitor may register with a special case positional placement indicator. However, additional system resources are used by the monitor dispatcher to satisfy special processing requirements of the monitor. Therefore, this option should be used only where necessary. To determine the need for this feature, refer to the individual device driver design for a detailed description of conditions under which it blocks.

DosMonReg issues a DosDevIOCtl (category 10, function 40h) monitor registration function call to the device driver. On receiving a monitor register request, a device driver must call the monitor dispatcher device helper to:

1. Create a monitor chain by calling the MonitorCreate device helper routine, if no monitor chain was previously created; and
2. Establish the monitor's input and output buffers within a monitor chain by calling the Register device helper routine.

Threads responsible for moving keystroke data through a monitor chain must pay special attention to the thread priority. Keystroke monitor threads must execute within the time critical priority class. More specifically these threads must execute at a priority level greater than or equal to the lowest level in the time critical priority class. The preferred level is level 0. The thread that makes the call to DosMonReg must also be executing in the time critical priority class.

Until there is a successful return from the DosMonReg call, no character will enter the monitor's input buffer. That is, the monitor will not have access to the device driver's data stream. The application is responsible for synchronizing completion of the DosMonReg call and the subsequent monitoring of the data stream with device input into the data stream. For an example and solution to this problem, see the "Type-ahead Characters" on page 8-26.

## DosMonRead

After an application has registered a pair of input and output buffers as a monitor for a device, it may take data from the data stream for filtering by calling DosMonRead. DosMonRead has three primary functions:

1. Waiting (optionally) for a signal from the monitor dispatcher that data has been placed into the monitor's input buffer
2. Moving that data from the monitor's input buffer into a private data area where it can be accessed freely by the monitor for filtering
3. Signaling the monitor dispatcher that data has been removed from the monitor's input buffer.

On calling DosMonRead, the application pushes the parameters onto the stack as shown in Table 8-3. The address parameters are *selector:offset* addresses of the following data areas within its address space:

- The input buffer previously registered with the monitor chain for the device
- A private data area into which the data record read from the input buffer is to be placed
- A Bytecnt variable that, on entry indicates the size of the private data area, and on exit indicates the size of the data record read from the input buffer.

In addition, the application can specify whether to block (wait) if no data is available in the monitor's input buffer.

Table 8-3 (Page 1 of 2). DosMonRead parameter stack

| **Parameter description** | **Size** |
|---|---|
| Address of monitor's input buffer | DWORD |
| Waitflag | WORD |

| Table 8-3 (Page 2 of 2). DosMonRead parameter stack | |
|---|---|
| **Parameter description** | **Size** |
| Address of private data area | DWORD |
| Address of Byte count variable | DWORD |

DosMonRead parameter validation includes verifying that:

1. The specified monitor input buffer is currently registered on an opened device handle.

   An application can DosMonRead only from an monitor input buffer already registered to an opened device. DosMonRead will return an error if the DosMonReg call registering the monitor's input and output buffers has not completed. That is, *a monitor does not have access to a device driver's data stream until DosMonReg completes successfully*.

2. The addresses of the buffers are valid.

   Null selectors passed as addresses for the monitor's input buffer or private data area will cause a GP fault to occur.

3. The monitor buffers and their contents have not been altered.

   If the monitor dispatcher detects that the monitor buffers have been corrupted or freed by the application, it prevents system failure by forcing a GP fault to occur. The application is abnormally terminated; and a pop-up message notifies the user that the process is being terminated.

4. The monitor is not terminating.

   After the monitor dispatcher begins monitor deregistration that is, the DeRegister device helper has been called by the character device driver, the monitor dispatcher prevents monitor threads from taking data from the data stream. A monitor thread calling DosMonRead after monitor deregistration begins receives an error. This guarantees minimal data loss during monitor deregistration when monitor buffers are removed from a monitor chain.

For each DosMonRead call, a *single* data record is taken from the monitor's input buffer and is placed into the monitor's private data area for filtering. A monitor data record is constructed by the device driver and consists of a flag WORD, which may be followed by actual device data. Monitor data records are variable length; that is, some may consist only of a flag WORD; some may consist of a flag WORD followed by actual device data of differing lengths. The maximum length of a monitor data record passing through a monitor chain is the length of the device driver's monitor chain buffer *minus 2 bytes*.

The device driver may use flags within the flag WORD to indicate the type of data that is part of this data record. The device driver may also use flags to indicate the action it expects its monitors to take when this data record is received. Device drivers may also have specific requirements for the return of data records to the data stream. Because of this, character device monitors should not indiscriminately consume data records from the data stream. Refer to the discussions in this book on the individual character device drivers for descriptions of their monitor records and expected actions.

Flushing a data stream is an important operation. At certain times it is necessary to guarantee that all data is removed from the data stream. At these times, a specially marked record, a *flush record*, is placed into the data stream by the device driver and must pass through all monitors in the chain. The flush data record consists of a single flag WORD, with the third bit of the first byte set. It is the only data record created by the monitor dispatcher and placed into a monitor chain by the monitor dispatcher on behalf of the device driver (see the description of the MonFlush device helper routine). When a flush record is received by a monitor, it must be returned to the data stream by calling DosMonWrite, after the monitor has taken the appropriate action. The action to be taken in response to the flush record will vary with the type of device and type of support required. To guarantee that all data has been removed from all buffers in the monitor chain, the monitor dispatcher prevents the device driver from placing additional data into the data stream until the flush record has passed through all monitors in the monitor chain. If the flush record is not returned to the data stream, the data stream will be severely and permanently impacted.

Because character device monitors are part of a data stream, they must process data records rapidly so that they do not delay I/O. A monitor application should be written so that the threads that actually read and write the monitor data run at a high priority. They should never perform operations such as I/O or semaphore waits that might delay them. The monitor application can have other threads running at normal priorities to handle such functions.

Threads responsible for moving keystroke data through a monitor chain must pay special attention to the thread priority. Keystroke monitor threads must execute within the time critical priority class. More specifically, these threads must execute at a priority level greater than or equal to the lowest level in the time critical priority class. The preferred level is level 0. This applies to any threads that read (DosMonRead), process, or write (DosMonWrite) keystroke monitor data.

Separate monitor threads from the same application may call DosMonRead to take data from the same monitor input buffer. To protect monitor input buffers during data movement, the monitor dispatcher guarantees that only one thread at a time from a single process may take data from a monitor's input buffer. The application, therefore, does not have to synchronize DosMonRead calls made by its separate threads.

If a monitor thread calls DosMonRead during monitor termination (for example, DosMonClose has been called from another thread belonging to the same application), it will receive an error return indicating that there is no data present in the monitor's input buffer. If a monitor thread is blocked on a DosMonRead call when monitor termination begins, it will be awakened and receive the same error return. In both cases, the monitor thread must handle the error return code. The application is, then, responsible for the orderly termination of its individual monitor threads.

## DosMonWrite

After an application has registered a pair of input and output buffers as a monitor for a device, it may return filtered data to the data stream by calling DosMonWrite. DosMonWrite has three primary functions:

1. Waiting for a signal from the monitor dispatcher that data has been removed from the monitor's output buffer, if there is not enough room in the monitor's output buffer for the data;
2. Moving filtered data from the monitor's private data area into the monitor's output buffer; and
3. Signaling the monitor dispatcher that data has been placed into the monitor's output buffer.

On calling DosMonWrite, the application pushes the parameters onto the stack as illustrated in Table 8-4. The address parameters are *selector:offset* addresses of the following data areas within its address space:

- The output buffer previously registered with the monitor chain for character device; and
- The private data area which contains the filtered data record to be returned to the data stream.

In addition, the application must specify a Bytecnt variable, indicating the size of the data record to be returned to its output buffer.

Table 8-4. DosMonWrite parameter stack

| **Parameter description** | **Size** |
|---|---|
| Address of monitor's output buffer | DWORD |
| Address of private data area | DWORD |
| Byte count | WORD |

DosMonWrite parameter validation includes verifying that:

1. The specified monitor output buffer is currently registered on an opened device handle.

   An application can DosMonWrite only into an monitor output buffer already registered to an opened device. DosMonWrite will return an error if the DosMonReg call registering the moni-

tor's input and output buffers has not completed. That is, *a monitor does not have access to a device driver's data stream until DosMonReg completes successfully*.

2. The addresses of the buffers are valid.

   Null selectors passed as addresses for the monitor's output buffer or private data area will cause a GP-fault to occur.

3. That the monitor buffers and their contents have not been altered.

   If the monitor dispatcher detects that the monitor buffers have been corrupted or freed by the application, it prevents system failure by forcing a GP fault to occur. The application is abnormally terminated; and a pop-up message notifies the user that the process is being terminated.

4. The monitor is not terminating.

   After the monitor dispatcher begins monitor deregistration that is, the DeRegister device helper has been called by the character device driver, the monitor dispatcher prevents monitor threads from returning data to the data stream. A monitor thread calling DosMonWrite after monitor deregistration begins receives an error. This guarantees minimal data loss during monitor deregistration when monitor buffers are removed from a monitor chain.

For each DosMonWrite call, a single filtered data record is placed into the monitor's output buffer. See the section above on the DosMonRead function call for a description of monitor data record format, flushing monitor buffers, and restrictions on data record consumption by a monitor.

Because character device monitors are part of a data stream, they must process data records rapidly so that they do not delay I/O. A monitor application should be written so that the threads that actually read and write the monitor data run at a high priority. They should never perform operations such as I/O or semaphore waits that might delay them. The monitor application can have other threads running at normal priorities to handle such functions.

Threads responsible for moving keystroke data through a monitor chain must pay special attention to the thread priority. Keystroke monitor threads must execute within the time critical priority class. More specifically, these threads must execute at a priority level greater than or equal to the lowest level in the time critical priority class. The preferred level is level 0. This applies to any threads that read (DosMonRead), process, or write (DosMonWrite) keystroke monitor data.

Separate monitor threads from the same application may call DosMonWrite to return filtered data to the same monitor output buffer. To protect monitor output buffers during data movement, the monitor dispatcher guarantees that only one thread at a time from a single process may return data to a monitor's output buffer. The application, therefore, does not have to synchronize DosMonWrite calls made by its separate threads.

If a monitor thread calls DosMonWrite during monitor termination (for example, DosMonClose has been called from another thread belonging to the same application), it will receive an error return indicating that it cannot return data to the monitor's output buffer. If a monitor thread is blocked on a DosMonWrite call when monitor termination begins, it will be awakened and receive the same error return. In both cases, the monitor thread must handle the error return code. The application is, then, responsible for the orderly termination of its individual monitor threads.

## DosMonClose

When an application no longer needs to monitor data passing through a character device it must call DosMonClose, to remove all monitor buffers that it has previously registered with the device.

On calling DosMonWrite, the application pushes the parameter onto the stack as illustrated in Table 8-5.

Table 8-5. DosMonClose parameter stack

| Parameter description | Size |
|---|---|
| Device handle | WORD |

DosMonClose verifies that the specified device handle is currently opened.

**Note:** If an application has freed the segment containing its monitor buffers before calling DosMonClose, the following will occur:

1. The application frees the segment containing its monitor buffers.

   The monitor dispatcher detects alteration of the monitor buffer segment and forces a general protection fault to occur. A pop-up message on the screen notifies the user that the application is being abnormally terminated. Termination of the monitor mechanism is initiated and the application's handle to the device is automatically closed.
2. The application calls DosMonClose before termination of the application has been initiated. Because the device handle is already closed, a TRAP occurs and another pop-up message, indicating that the process is stopped, is displayed on the screen.

DosMonClose calls on the OS/2 File System to close the application's device handle for monitors by calling DosClose. The file system, in turn, sends a monitor close request to the device driver . At this time, the device driver must call on the monitor dispatcher to remove all monitor buffers registered by the application by calling the DeRegister device helper routine for each of its monitor chains. (The application may have registered monitor buffers with more than one monitor chain for the same device.) On the return from each DeRegister call, if the monitor chain is empty (that is, there are no more monitors in the monitor chain) and the device driver no longer needs to use it, the device driver may (optionally) issue the MonitorCreate device helper call with the DELETE option to remove the monitor chain.

**Note:** The device driver must receive notification that a monitor process is terminating so that the monitor dispatcher can be called to remove the monitor buffers from the monitor chain. The Monitor Dispatcher has this responsibility to guarantee an orderly removal of the monitor from the monitor chain with minimal data loss, while maintaining the integrity of data movement through other monitors in the monitor chain.

The OS/2 File System is already aware that a monitor has opened a handle to the device during a previous DosMonOpen call. When a monitor process terminates (normally or abnormally) all handles opened by that process are closed. The file system notifies the device driver that monitor termination is occurring by sending a monitor close request to the device driver.

In the event that an application calls DosExit to terminate the process without calling DosMonClose, the file system sends a monitor close request to the device driver so that the monitor dispatcher can remove the monitor buffers from all monitor chains.

# Guidelines for a Character Device Monitor

The following is a summary of requirements and restrictions for character device monitors.

## Monitor Buffers

Applications allocate their own monitor input and output buffers. A monitor input/output buffer pair must be allocated from the same segment. Before registering the monitor buffers with a monitor chain belonging to a character device driver, the application is required to provide the size of the buffer in BYTES in the first WORD of each buffer, length WORD inclusive.

The OS/2 monitor dispatcher manages all monitor buffers registered by applications. During monitor registration, the monitor dispatcher initializes each buffer being registered. After monitor registration, an application should not touch its monitor buffers, that is, directly access any portion of the buffers. The monitor dispatcher directly accesses these buffers on behalf of the application when the application calls DosMonRead and DosMonWrite.

Although a monitor application should not touch its monitor input and output buffers after monitor registration, the monitor dispatcher cannot prevent this from happening. Corruption of the monitor input and output buffers by the application will result in termination of the application.

More than one application can monitor the same data stream. Monitors belonging to separate applications may have input and output buffers of various sizes. To guarantee data movement through all buffers in a monitor chain, therefore, the monitor dispatcher defines a *minimum monitor buffer length*. The minimum buffer length is defined as the length of the character device driver's monitor

chain buffer plus 20 BYTES. The length of a monitor's input and output buffers must be at least this length. This is also the recommended length of the private data area specified on a DosMonRead call.

Because a monitor chain buffer is defined and owned by a character device driver, its length is documented in the descriptions of the individual device drivers that support monitors. See the sections in this book describing the keyboard, mouse, and printer device drivers for this information.

## Monitor Data Records

Only one monitor data record at a time can be taken from a data stream by calling DosMonRead or returned to the data stream by calling DosMonWrite. A monitor data record is constructed by the device driver and consists of a flag WORD, which may be followed by actual device data. Monitor data records are variable length; that is, some may consist solely of a flag WORD; some may consist of a flag WORD followed by actual device data of differing lengths. To be consistent with the minimum buffer size defined for a monitor chain, the maximum length of a monitor data record passing through a monitor chain is the length of the device driver's monitor chain buffer minus 2 BYTES. An application cannot return to its output buffer a data record larger than this.

The device driver may use flags within the flag WORD to indicate the type of data that is part of this data record and/or the action it expects its monitor to take on receiving this data record. Device drivers may have specific requirements for the return of data records to the data stream. Therefore, monitors should not indiscriminately consume data records from the data stream. Refer to the discussions on the individual device drivers in this book for descriptions of their monitor records and expected actions.

Flushing a data stream is an important operation. At certain times it is necessary to guarantee that all data is removed from the data stream. At these times, a specially marked record, a *flush record*, is placed into the data stream by the device driver and must pass through all monitors in the chain. The flush data record consists of a single flag WORD, with the third bit of the first byte set. It is the only data record created by the monitor dispatcher and placed into a monitor chain by the monitor dispatcher on behalf of the device driver (see the description of the MonFlush device helper routine). When a flush record is received by a monitor, it must be returned to the data stream by calling DosMonWrite, after the monitor has taken the appropriate action. The action to be taken in response to the flush record will vary with the type of device and type of support required. To guarantee that all data has been removed from all buffers in the monitor chain, the monitor dispatcher prevents the device driver from placing additional data into the data stream until the flush record has passed through all monitors in the monitor chain. If the flush record is not returned to the data stream, the data stream will be severely and permanently impacted.

## Positioning of Monitors in a Monitor Chain

Monitors are registered on a monitor chain with a positional preference parameter:

- 0 = *DEFAULT*.
- 1 = *FIRST*;
- 2 = *LAST*; or

Monitor buffers are placed in a monitor chain in a position relative to monitors already registered with the monitor chain.

The first monitor in a chain registered as *FIRST* will be at the head of the monitor chain. The next monitor registered as *FIRST* will follow the first monitor registered as *FIRST*, and so forth.

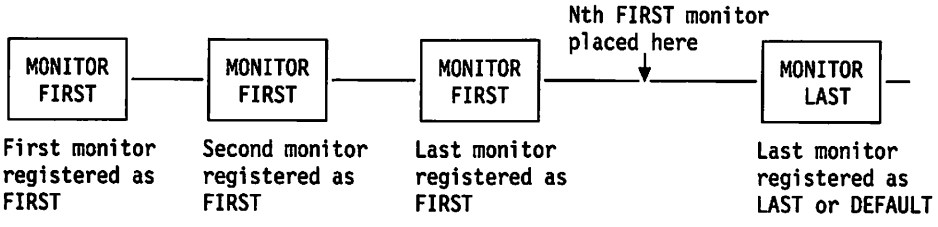



Figure 8-7. Monitors registered as FIRST

Similarly, the first monitor registered as *LAST* will be at the very end of the monitor chain. The next monitor registered as *LAST* will precede the first monitor registered as *LAST*, and so forth.

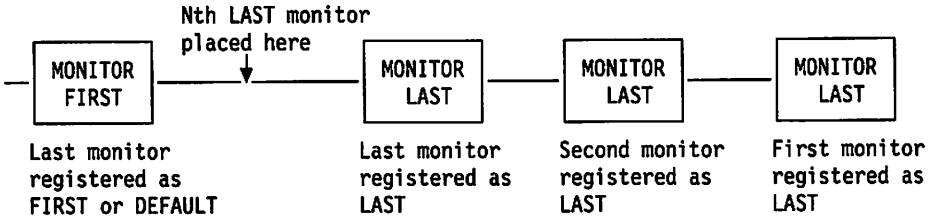



Figure 8-8. Monitors registered as LAST

The first monitor registered as *DEFAULT* will precede the last monitor registered as *LAST*. The next monitor registered as *DEFAULT* will precede the first monitor registered as *DEFAULT*.

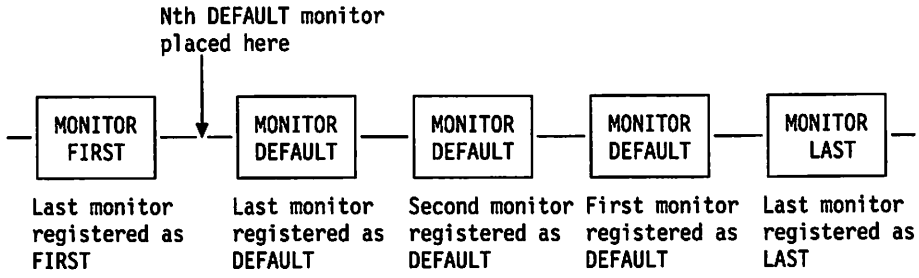



Figure 8-9. Monitors registered in a DEFAULT position

**Note:** It is possible for the last monitor registered as *LAST* or the last monitor registered as *DEFAULT* to be the first monitor in the chain.

## Monitor Thread Priorities

To guarantee steady and reasonably fast data throughput in a monitor chain in this multi-threaded, multi-tasking environment, the priority of monitor threads must be set high.

Threads responsible for moving keystroke data through a monitor chain must pay special attention to the thread priority. Keystroke monitor threads must execute within the time critical priority class. More specifically, these threads must execute at a priority level greater than or equal to the lowest level in the time critical priority class. The preferred level is level 0. This applies to any threads that read (DosMonRead), process, or write (DosMonWrite) keystroke monitor data. In addition, the thread that makes the call to DosMonReg must also be executing in the time critical priority class.

It is generally recommended that a monitor application call DosCreateThread to create separate threads to intercept data from and return filtered data to a monitor chain. The application also must set the priority of these threads by calling DosSetPrty. See the Programmer's Reference Library for the descriptions of the DosCreateThread and DosSetPrty function calls, for more information concerning creating threads and priority classes, respectively.

## Special Considerations for Character Device Monitors

The following section summarizes conditions to be considered when writing a character device monitor.

### Performance

Several factors can affect the performance of a character device monitor, that is, its ability to intercept, filter and return data to a data stream quickly and efficiently. These include:

•*Separate threads*

Because character device monitors are part of a data stream, they must process data records rapidly so that they do not delay I/O. A monitor application should be written so that the threads that actually read and write the monitor data run at a high priority. They should never perform operations such as I/O or semaphore waits that might delay them. The monitor application can have other threads running at normal priorities to handle such functions.

•*Task synchronization*

Semaphores may be used to synchronize and serialize the separate tasks of taking data from the data stream, filtering the data and returning the filtered data to the data stream. For example, a monitor thread that has completed a DosMonRead call may signal another monitor thread that it can filter the data. This second thread, having filtered the data, may signal a third monitor thread that the filtered data can be returned to the data stream by calling DosMonWrite. The third thread, having called DosMonWrite, may signal the first thread that it can take more data from the data stream. To avoid impacting the data stream severely or permanently, however, the application must take special care that its monitor threads do not wait on semaphores cleared by outside events, for example, waiting for I/O.

•*I/O requests*

Separate, non-monitor threads within a monitor application may make I/O requests of other devices which will not delay movement of data through the monitor. However, a monitor thread intercepting, filtering, and returning data to the data stream should not make I/O requests of the device whose data stream it is monitoring. Such a request will result in a deadlock; data movement through the data stream will be permanently suspended. For example, a thread belonging to a keystroke monitor application calls DosMonRead, filters the data, and calls DosMonWrite. Before calling DosMonWrite, however, the thread makes a KBDCHARIN keyboard subsystem call to take data from the keyboard API buffer. Because no data has reached the keyboard device driver's monitor chain buffer and, therefore, the keyboard API buffer, the monitor thread blocks. The monitor thread cannot return the filtered data to the data stream, and therefore to the device driver's monitor chain buffer.

•Data consumption

To prevent other deadlocks, a character device monitor must follow the rules set by the character device driver and monitor dispatcher on data record consumption. The monitor dispatcher requires flush records to be returned to the data stream. Device drivers may have specific requirements for the return of data records to the data stream. For example, a character device driver may place a data record into a monitor chain and then block until that data record is returned from the monitors into its monitor chain buffer. If a monitor removes that data from the data stream by calling DosMonRead and does not return that data to the data stream by calling DosMonWrite, it will never reach the device driver's monitor chain buffer. The device driver will never unblock its blocked condition. Because of this, monitors should not indiscriminately consume data records from the data stream.

•*Expected responses*

The device driver may use flags with the flag WORD of a data record to indicate the type of data that is part of this data record, and the action the device driver expects its monitor to take when it receives this data record. The monitor may be also be expected to indicate a response to the action requested by the device driver by changing the flags before returning the data record to the data stream. In this way, flags in a monitor data record are used in a dialog between the device driver

and its monitors. Failure on the part of a monitor to respond as expected to a request from the device driver may result in a deadlock.

•*Error handling*

The developer of a character device monitor should not disregard error returns from monitor function calls and assume that *good data* is automatically received. A character device monitor must handle errors returned from all monitor function calls. Failure to do so can result in severe and permanent impacting of the data stream. For example, a monitor thread that ignores an error returned on a DosMonRead call may attempt to DosMonWrite whatever data that is currently residing in the monitor's private data area. This data may be meaningful, that is, it may be the good data returned from the last DosMonRead call. This data may not be meaningful, but may be the result of some other internal processing by the application. In either case, data unexpected by the device driver may be returned to the data stream.

## Special Processing Requirements

The monitor dispatcher places filtered data, that is, data that has passed through all monitors in a monitor chain, into the device driver's monitor chain buffer. The monitor dispatcher signals the device driver that data is there by calling a notification routine within the device driver. The device driver must process this data and return to the monitor dispatcher so that data can continue to flow within the monitor chain. If the device driver cannot process the data and BLOCKS on the notification routine call, monitors in the monitor chain will receive no more data. See "Data Passing through a Monitor Chain" on page 8-9.

For the printer, the device may be disabled, or turned off. Spooling stops. For the keyboard, applications may not be requesting data from the keyboard buffer, even though keystrokes are still being entered. The keyboard device driver will generate a series of tones on the speaker, indicating that it cannot process more keystrokes.

An application, however, may require that it continue receiving data from the data stream even though the device driver, or another monitor *downstream* in the monitor chain, has stopped processing filtered data. The application may indicate this requirement by registering its monitor with the *special case* positional placement indicators. Refer to the description of the DosMonReg function call in the Programming Reference Library for details.

The OS/2 Spooler is an example of a monitor with this processing requirement. When the printer device driver cannot send filtered data to the printer device because the device is disabled, the spooler should continue receiving data and creating spool files on disk. Because the device driver will block, the spooler must register its monitors with the special case positional placement indicator so that the Monitor Dispatcher can satisfy its special processing requirement. The Spooler will continue to receive data as long as:

- There is data available; and
- The spooler continues to process the data it receives by calling DosMonRead (that is, as long as it can continue to save the spool data on disk).

**Note:** Any monitor may register with a special case positional placement indicator. However, because additional system resources are used by the monitor dispatcher to satisfy special processing requirements of the monitor, this option should be used only where necessary. To determine the need for this feature, refer to the discussions in this book on the individual device drivers for a detailed description of conditions under which they block.

## Monitor Termination

An application is responsible for terminating its own monitor threads. The monitor dispatcher is responsible for removing the monitor's buffers from the monitor chains belonging to a character device driver. Removal of the monitor buffers from the monitor chains is initiated by the character device driver when it receives a *monitor close* request. This occurs when:

- DosMonClose is called by the application.
- DosExit (terminate all threads in a process) is called by the application.
- The process is abnormally terminated by the user (for example, he presses CTRL-C key sequence).

- The process is abnormally terminated by the monitor dispatcher when it detects that the application's monitor buffers have been altered (for example, the monitor buffers have been corrupted by the application).

When the character device driver receives a monitor close request, it calls the DeRegister device helper routine for each of its monitor chains. To protect the integrity and order of data in the data stream and guarantee minimal data loss, the monitor dispatcher:

- Locks out the application from subsequent DosMonRead and DosMonWrite calls

  The application is prevented from intercepting data from and returning data to the data stream

- Drains all data in an orderly manner from the data stream

before returning to the device driver.

The successful removal of monitor buffers from a monitor chain is dependent on the ability of the device driver or other monitors downstream in the monitor chain to receive and process data from the data stream. If the monitor being removed from the monitor chain is not the last in the monitor chain, the next monitor in the chain must not be blocked. Similarly, the device driver must NOT be blocked on a call to its notification routine; that is, it must be able to process data it receives in its monitor chain buffer.

During monitor termination, the monitor dispatcher prevents the application from touching the data stream by returning errors on DosMonRead or DosMonWrite calls. In addition, monitor threads blocked on a DosMonRead or DosMonWrite call are awakened and the calls return errors. The application can then clean up the appropriate monitor threads. In both cases, the application is responsible for checking and handling the error returns.

During monitor termination there is always a risk of data loss. When the lost data is critical data on which the device driver or monitor dispatcher is blocked, the system may be permanently impacted. To reduce this risk, a monitor application should be *well-behaved*. See "Well-behaved Monitor Applications." Separate threads monitoring the data stream should be terminated before DosMonClose is called. This guarantees that all data taken from the data stream is returned to the data stream by the application. The application should stop calling DosMonRead and DosMonWrite before it calls DosMonClose. This may mean synchronizing the calls with semaphores.

The closedown of monitor threads can also be synchronized by using *signal handler* or *exitlist* routines. Signal handler routines are called when the system sends a signal to the process that certain events are occurring, for example, the CTRL-C key sequence has been pressed. Exitlist routines are a list of routines that are called when an application is terminating. In either case, these routines can be used to signal an application's separate monitor threads to prepare for termination. See the descriptions of the DosSetSigHandler, DosHoldSignal, and DosExitList function calls in the the Programming Reference Library for details.

## Well-behaved Monitor Applications

So that data loss is minimal and system performance is not impacted, a monitor application should be *well-behaved*. The following characteristics are typical of well-behaved monitor applications:

1. Having separate threads intercepting data from the data stream (calling DosMonRead) and returning filtered data to the data stream (calling DosMonWrite)
2. Following the device monitor *rules*, as defined by the monitor dispatcher and the individual character device driver whose data stream is being monitored

   This includes definition of data streams, format of data records passing through the monitor chain, flags and expected actions, and consumption of data records.
3. Having error handling procedures for each function call
4. Using semaphores and shared memory carefully so that blockages do not occur because of the monitor application, thereby blocking the entire data stream
5. Terminating data monitoring before calling DosMonClose, that is, all DosMonRead and DosMonWrite threads issuing no more calls before DosMonClose

6. Including signal handler or exitlist routines to coordinate the closedown of monitor threads in an orderly manner.

A well-behaved, multi-threaded character device monitor can be represented by the pseudocode in the figures which follow:

```
Application's DATA

  Define Device-Specific Data:

    Device name string      (8 ASCII characters, with nul at end)
    Index                   (indicating data stream to be monitored)
    Length of monitor chain buffer
     (minimum buffer size for all monitor buffers in monitor chain)

  Define Monitor Data Areas:

    (The size of each of these areas is based on the size of the
     device driver's monitor chain buffer for the data stream
     we will be monitoring.)

    Monitor Input Buffer
    Monitor Output Buffer
    Private Data Area

  Define additional Monitor Data:

    Device handle               (returned from DosMonOpen call)
    Byte count variable                 (size of data record
          received by DosMonRead and returned by DosMonWrite)
    Wait flag               (do we want to wait for DosMonRead?)

  Define a RAM semaphore to synchronize monitor threads:

    Semaphore A

  Define a variable to track error returns from DosMonRead and
    DosMonWrite:

    SetError
```

Figure 8-10. A well-behaved monitor application's DATA definitions

```
Application's MAIN routine

  CALL DosMonOpen       to open the device.

  CALL DosSetPrty       to set the priorities of our threads high.

  CALL DosMonReg        to register the monitor input and output
                        buffers with the data stream.

  CALL DosSemSet        to set RAM Semaphore A, which will be used to
                        synchronize termination of the application.

  CALL DosCreateThread  to create a separate thread that executes
                        the application's MONITOR routine.

  CALL DosSemWait       to wait for RAM Semaphore A to be cleared
                        by the application's child thread executing
                        the MONITOR routine.

  CALL DosMonClose      to terminate the monitor and close the device
                        handle.

  CALL DosExit          to terminate the application.
```

Figure 8-11. A well-behaved monitor application's MAIN routine

```
Application's MONITOR routine

  WHILE SetError = 0  the application's CHILD thread will execute as
                      long as there are no errors returned from
                      DosMonRead or DosMonWrite calls.

    CALL DosMonRead   to take data from the monitor's input buffer,
                      that is, from the data stream.

    SetError = error returned from DosMonRead

    IF SetError = 0 THEN  no error was returned from DosMonRead and
                          we can continue.

      Filter the data.

      CALL DosMonWrite  to return the filtered data to the monitor's
                        output buffer, that is to the data stream.

      SetError = error return from DosMonWrite

    ENDIF

  END WHILE

  CALL DosSemClear      to clear RAM Semaphore A, to signal parent
                        thread that we are done monitoring the
                        data stream.

  CALL DosExit          to terminate only this thread.
```

Figure 8-12. Application's MONITOR routine

## Monitor Problems and Solutions

The checklist in Table 8-6 describes symptoms that may be experienced in a monitor environment and suggests the problems that cause them.

Table 8-6. Character device monitor checklist

| Symptom | Problem |
|---|---|
| DosMonReg returns an invalid parms error. | Could be trouble with monitor buffers:<br>1. Are they in the same segment?<br>2. Are they at least as large as the device driver's monitor chain buffer plus 20 BYTES?<br>3. Does the first WORD of each buffer contain the length of the buffer in BYTES? |
| Monitor does not appear to be running. It is not receiving data from DosMonRead calls or returning data on DosMonWrite calls. | Are your monitor threads running at a high priority? |
| Performance is poor. Data moving slowly through monitors. | Is your application doing complex processing in threads separate from your monitor (DosMonRead/DosMonWrite) threads? |
| Filtered data is not returned to the device driver. | Is your monitor consuming data records (that is, not returning them to the data stream by calling DosMonWrite)? If the device driver requires that these data records be returned, your monitor is blocking the data stream. |
| The system has stopped running. | Are your monitor or non-monitor threads polling the device's API buffer? Your monitor threads may be unable to return data to the data stream. The data, therefore, will never reach the device driver's API buffer. Applications will wait forever for data from READs. |

The following section describes special monitor problems and suggests solutions.

### Type-ahead Characters

An application may require monitoring all characters passing through a character device. The application also may require that *type-ahead characters* are intercepted. Type-ahead characters are those keystrokes entered by a user after starting a program and before prompts for information are given. Monitor applications must do some additional work to intercept these keystrokes.

There is a time *window* between the time when an application is invoked and the time when the registration of the application's monitor is completed. During this time window, the application does not have access to the data stream. Because the monitor is not yet registered, it cannot intercept keystrokes. The device driver processes type-ahead keystrokes, placing them into its API buffer.

**Note:** If other applications have monitors registered with the monitor chain associated with the same data stream, type-ahead keystrokes will be filtered by these applications and returned to the device driver for processing. Registration of other monitor applications with the same monitor chain is irrelevant.

To monitor type-ahead characters, an application must:

1. Register its monitor
2. Read and save type-ahead characters from the API buffer after monitor registration is completed and *before calling DosMonRead*
3. Play back the saved type-ahead characters and return them to the data stream by calling DosMonWrite
4. Monitor the data stream as usual, intercepting data from the data stream by calling DosMonRead and returning filtered data to the data stream by calling DosMonWrite.

Pseudocode for the solution to this problem is illustrated in Figure 8-13 on page 8-28.

Our Application

Description:
- requires monitoring all characters for a device, including type-ahead characters.
- a single-threaded application.

CALL DosMonOpen to open a handle to the device.

CALL DosSetPrty to set the thread priority high.

CALL DosMonReg to register a monitor with the device.

Before we start intercepting data, we want to get and save any type-ahead data that the device driver has processed before DosMonReg returned.

WHILE the device driver's API buffer is not empty

    Get a character from the API buffer, using the appropriate device subsystem call.

    Save the characters in a temporary buffer.

END WHILE

Play back the type-ahead characters

WHILE our temporary buffer is not empty

    Get a character from the temporary buffer.

    Build a monitor record, according to the specifications of the device driver.

    Filter the data.

    CALL DosMonWrite to return it to the data stream and, therefore, to the device driver. The device driver processes the data by placing it into its API buffer so that it can be received and processed by an application.

END WHILE

Monitor the data stream as usual.

Figure 8-13. Monitoring type-ahead keystrokes

```
WHILE we want to monitor the data stream

    CALL DosMonRead   to get data from the data stream.

    Filter the data.

    CALL DosMonWrite  to return the filtered data to the data stream.

END WHILE

CALL DosMonClose      to remove the monitor buffers from the monitor
                      chain and close the handle to the device.

CALL DosExit          to terminate this application.
```

Figure 8-14. Monitoring type-ahead keystrokes

## Redirecting Data to Another Device

Character device monitors can be used to redirect data from one character device to another, as long the corresponding device drivers provide monitor support. This is done by:

1. Opening both devices
2. Registering a monitor with the first device
3. Registering a monitor with the second device
4. Intercepting data from the data stream belonging to the first device by calling DosMonRead
5. Using the data taken from the data stream belonging to the first device and creating a monitor record in the format defined by the second device driver
6. Placing the new data record into the data stream belonging to the second device by calling DosMonWrite.

The following is a pseudocode example of how keystroke data can be redirected to the printer.

CALL DosMonOpen to open keyboard device.

CALL DosMonOpen to open printer device.

CALL DosSetPrty to set priority of thread high.

CALL DosMonReg to register a set of buffers as a keystroke monitor
for the data stream associated with the current session.

CALL DosMonReg to register another set of buffers as a printer monitor
for the printer device's 'data' monitor chain.

WHILE we want to continue redirecting data

CALL DosMonRead to read a data record from the keystroke monitor's
input buffer.

CALL DosMonWrite to return the keystroke monitor data record to the
keyboard device driver. (Some data, for example
FLUSH records, MUST be returned to the keyboard data
stream.)

Translate the keystroke monitor data record into a printer monitor
data record.

CALL DosMonWrite to place the printer monitor data record into the
printer monitor's output buffer, thereby sending
the data to the printer device.

END WHILE

CALL DosMonClose to close the handle to the printer device.

CALL DosMonClose to close the handle to the keyboard device.

CALL DosExit to terminate the application.

Figure 8-15. Redirecting keystroke data to the printer

**Note:** Device drivers, as well as the monitor dispatcher, expect the return of certain monitor data records from its monitors. This requires that certain data not only be redirected to another data stream but also be returned to its original data stream. See the sections in this book describing the individual device drivers for information on restrictions for consumption of monitor data records.

The advantage of this technique to redirect data from one device to another is that it can be done without modifying or rewriting a character device driver. The disadvantage of using this technique is reduced performance. When performance is critical, data should be redirected between device drivers by using the inter-device driver communication mechanism provided by OS/2.

# Providing Monitor Support in a Character Device Driver

## The Monitor Dispatcher Device Helper

Character device drivers provide support for character device monitors by using the Monitor Dispatcher Device Helper routines to:

1. Create monitor chains for their data streams (MonitorCreate)
2. Register monitor buffers with their monitor chains on behalf of monitor applications (Register)
3. Send data to their monitors (MonWrite)
4. Flush all data from their monitors (MonFlush)
5. Remove monitors buffers from their monitor chains on behalf of monitor applications (DeRegister).

Character device drivers use the Monitor Dispatcher Device Helper routines to manage the monitor chains associated with their data streams and to move data between different monitors in their monitor chains.

The Monitor Dispatcher Device Helper routines are part of the OS/2 monitor dispatcher. All OS/2 device helper routines are part of the OS/2 kernel and are loaded at the most privileged level (0). Device Helper services are invoked by passing parameters through registers, by loading a function code into the DL register and making a FAR call to the DevHlp interface routine. The AX register is used to return errors. The Carry Flag is clear, if no errors are returned, and is set, if an error is returned. Refer to the descriptions of the individual device helper routines in the Device Helper Services chapter of this book.

Each OS/2 monitor dispatcher device helper routine uses the OS/2 System Trace Facility. Entry and exit parameters are traced when tracing is enabled for monitors. This facility provides assistance for the character device driver developer during the debug phase of a device driver's development. Refer to the Programming Guide for a description of the System Trace Facility.

### MonitorCreate

As previously stated in this chapter's introduction, individual device drivers determine how their data streams are defined. For example, the keyboard and mouse device drivers define data streams for each OS/2 session. The printer device driver define data streams for each printer device. Each data stream may be monitored by a *chain* of one or more monitors. Before monitor registration can occur, however, the device driver must create a monitor chain for its data stream by calling the MonitorCreate device helper routine.

The MonitorCreate device helper routine can be called in protect mode only, at task time or at Init time. The device driver may call the MonitorCreate device helper anytime prior to registering a monitor; when:

1. The device driver is installed.

   The device driver developer may prefer to create all monitor chains for each data stream during initialization of the device driver. Monitor chains are created regardless of their usage. For example, a character device driver may create monitor chains for each OS/2 session during initialization. Because each monitor chain requires additional system resources, this can be a resource-expensive choice when monitors may not likely be registered on the monitor chains. The developer should consider this when designing and developing the device driver.

2. The device driver receives a monitor open request.
3. The device driver receives a monitor register request (when no monitor chain has been created, and before the Register device helper routine is called to register the monitor).

   The device driver developer may prefer to create all monitor chains as they are needed. For example, a data stream may be defined when an OS/2 session is started. The monitor chains associated with that data stream may be created only when an application calls DosMonReg to register a monitor.

On calling the MonitorCreate device helper routine, the device driver places the parameters to the call in the registers as illustrated in Table 8-7 on page 8-32. The parameters include the selector:offset addresses of the device driver's:

1. *Monitor chain buffer*

   This buffer is the last buffer in a monitor chain and must reside within the device driver's first DATA segment (that is, the device driver's header segment). The monitor dispatcher places filtered data (that is, data that has passed through all monitors in the monitor chain) into this buffer. The device driver processes this data by placing it into its API buffer or by sending it to the device.

   **Note:** On calling the MonitorCreate device helper routine, the device driver must specify the LENGTH of the device driver's monitor chain buffer within the first WORD of the buffer, length WORD inclusive. To guarantee data movement through all monitor buffers in the monitor chain, the monitor dispatcher defines the *minimum buffer size for all monitor buffers in the monitor chain* as the length of the device driver's monitor chain buffer plus 20 bytes (reserved for buffer management by the Monitor Dispatcher). Similarly, data records passing through a monitor chain will not exceed the length of the device driver's monitor chain buffer minus 2 BYTES.

   **Note:** The monitor dispatcher places filtered data into this buffer, starting at the second WORD of the buffer. The monitor dispatcher places the length of the data (in BYTES) in the first WORD of the buffer, overwriting the original length word for the buffer.

2. *Notification routine.*

   This routine is called by the monitor dispatcher when filtered data is placed into the device driver's monitor chain buffer. This routine must reside in the device driver's first CODE segment.

In addition, the device driver specifies a monitor chain handle parameter. This parameter is used to indicate creation or deletion of a monitor chain. If the handle is zero when MonitorCreate is called, a new monitor chain is created and the *monitor chain handle* is returned to the device driver. If the handle is non-zero when MonitorCreate is called, the monitor chain with that handle is deleted.

Table 8-7. Register usage for Monitorcreate

| **Registers** | **Parameter Description** |
|---|---|
| ES:SI | Address of monitor chain buffer |
| DS:DI | Address of notification routine |
| AX | = 0 to create a new monitor chain (returns handle)<br>< >0 to delete a specific monitor chain |

The monitor chain handle is known only to the device driver and to the monitor dispatcher. The device driver uses this handle to identify the monitor chain to the monitor dispatcher on subsequent Monitor Dispatcher Device Helper calls. For example, when the device driver registers a monitor on behalf of an application with its monitor chain, it uses the monitor chain handle returned from a previous MonitorCreate call to indicate on which monitor chain the monitor should be registered.

The device driver maintains the list of monitor chain handles for its monitor chains, as well as the number of monitors registered on each monitor chain. When there are no monitors registered with a monitor chain, the monitor chain is *empty*.

**Note:** Monitor chains are initially created *empty*.

An empty monitor chain may be deleted by the device driver by calling the MonitorCreate device helper routine, with the handle parameter set to the handle of the empty monitor chain. Device drivers that create their monitor chains as they are needed generally delete empty monitor chains when they are no longer needed.

## Register

When an application calls DosMonReg to register a monitor with a character device driver, the device driver receives an IOCTL monitor register request. Based on the INDEX parameter to DosMonReg, the device driver decides:

1. Which data stream the application is requesting access to
2. Which monitor chain the monitor should be registered on.

If no monitor chain has been defined or created for the data stream indicated, the device driver must first call the MonitorCreate device helper routine to create one. The device driver then calls the Register device helper routine to instruct the Monitor Dispatcher to add the monitor buffers to the monitor chain.

The Register device helper routine can be called in protect mode only, at task time. On calling the Register device helper routine, the device driver places the parameters to the call in the registers as illustrated in Table 8-8. The parameters include:

1. The selector:offset address of the monitor's input and output buffers
2. A flag indicating positional placement in the monitor chain (see "Positioning of Monitors in a Monitor Chain" on page 8-19 and "Special Processing Requirements" on page 8-22)
3. The process ID (PID) of the application requesting monitor registration
4. The handle of the monitor chain on which it is being registered.

Table 8-8. Register usage for Register

| Registers | Parameter Description |
|---|---|
| ES | Selector of segment containing monitor's buffers |
| SI | Offset of monitor's input buffer |
| DI | Offset of monitor's output buffer |
| CX | monitor's PID |
| DH | positional placement indicator |
| AX | monitor chain handle |

## MonWrite

A character device driver places data into a monitor chain associated with one of its data streams by calling the MonWrite device helper routine. If monitors are registered with the monitor chain (that is, the monitor chain is not empty), data placed into the monitor chain will be automatically moved by the monitor dispatcher into the monitor input buffer belonging to the monitor in the chain. If monitors are not registered with the monitor chain (that is, the monitor chain is empty), data placed into the monitor chain will be automatically moved by the monitor dispatcher into the device driver's monitor chain buffer.

**Note:** The device driver's notification routine is called when this occurs.

The MonWrite device helper routine can be called in protect and real mode, at task time or at interrupt time. On calling the MonWrite device helper routine, the device driver places the parameters to the call in the registers as illustrated in Table 8-9 on page 8-34. The parameters include:

1. The selector:offset address of the data record to be placed into the monitor chain

   This address must be located within the device driver's first DATA segment (that is, the device driver's header segment).
2. The size of the data record (number of BYTES)
3. The handle of the monitor chain associated with the data stream.

In addition, the device driver may specify whether the monitor dispatcher should block or wait until it can place data into the monitor chain.

| Table 8-9. Register usage for MonWrite | |
|---|---|
| **Registers** | **Parameter Description** |
| DS:SI | Address of data record to send to monitors |
| CX | Size of data record (bytes) |
| AX | device driver's handle for monitor chain |
| DH | Waitflag |

A device driver may call the MonWrite device helper routine at interrupt time. If the device driver specifies the option to block (wait until data is successfully placed into the monitor chain) at interrupt time, the device driver will receive an error from the monitor dispatcher if it is unable to place the data into the monitor chain. So that no data is lost, the device driver should reissue the call.

To prevent interruption while moving data during this call, the monitor dispatcher disables interrupts. This can be critical to the performance of a device driver that calls MonWrite to write data into an empty monitor chain. Under these circumstances, the monitor dispatcher:

1. Disables interrupts
2. Places the data into the device driver's monitor chain buffer
3. Calls the device driver's notification routine
4. Enables interrupts when the device driver returns from its notification routine.

When the device driver receives data in its monitor chain buffer, it must process that data quickly (placing it into its API buffer or sending it to its device). The time between the call to the notification routine and the return to the monitor dispatcher must be minimal so that interrupts are not disabled too long.

## MonFlush

A character device driver sometimes requires that all data placed into a monitor chain has passed through all monitors in the chain and has been returned to the device driver. A character device driver may call on the monitor dispatcher to *flush* all buffers in a monitor chain, that is, remove all data from all monitor buffers by calling the MonFlush device helper routine.

The MonFlush device helper routine can be called in protect mode only, at task time. On calling the MonFlush device helper routine, the device driver places the parameters to the call in the registers as illustrated in Table 8-10. The device driver specifies the handle of the monitor chain to be flushed.

| Table 8-10. Register usage for MonFlush | |
|---|---|
| **Registers** | **Parameter Description** |
| AX | device driver's handle for monitor chain |

The monitor dispatcher places a *flush record* (a single WORD data record with the third bit of the first BYTE set) into the monitor chain. To guarantee that all data has been removed from all buffers in the monitor chain, the monitor dispatcher prevents the device driver from placing additional data into the data stream until the flush record has passed through all monitors in the monitor chain. If the flush record is not returned to the data stream, the data stream will be severely and permanently impacted.

**Note:** Because placement of data into the monitor chain is suspended while a flush record passes through a monitor chain, *monitors must not consume flush records*. Flush records obtained from the data stream from a DosMonRead call must be returned to the data stream on a DosMonWrite call.

The action to be taken by a monitor application in response to the flush record will vary with the type of device and type of support required. Refer to the discussions in this book on the individual character device drivers for descriptions of their monitor records and expected actions.

## DeRegister

When a monitor application terminates normally (by calling DosMonClose and DosMonExit) or abnormally (for example, CTRL-C has been pressed or the monitor buffers have been corrupted), the file system sends a monitor close request to the character device driver. When the device driver receives this request, it must call on the monitor dispatcher to remove all monitors registered by the terminating process from all monitor chains belonging to the device driver. It calls the DeRegister device helper routine *for each non-empty chain*, that is, for each monitor chain for which there are monitors registered.

The DeRegister device helper routine can be called in protect mode only, at task time. On calling the DeRegister device helper routine, the device driver places the parameters to the call in the registers as illustrated in Table 8-11. For each DeRegister device helper call, the device driver provides the monitor dispatcher with the PID (process ID) of the terminating application and the handle of a non-empty monitor chain.

Table 8-11. Register usage for DeRegister

| Registers | Parameter Description |
|---|---|
| AX | device driver's handle for monitor chain |
| BX | PID of monitor process |

For each DeRegister call, the monitor dispatcher removes all monitors registered by the terminating process from the specified monitor chain and returns to the device driver the number of monitors still registered with the chain.

During the DeRegister call, the monitor dispatcher reduces the risk for data loss by:

1. Suspending subsequent MonWrite and MonFlush device helper calls to place data into the monitor chain until the monitor buffers belonging to the terminating application have been removed from the monitor chain
2. Marking the monitor's input and output buffers as closed to subsequent data movement through DosMonReads and DosMonWrites
3. Draining all data from the monitor's input and output buffer in an orderly manner.

So that all data can be drained from the monitor's buffers, the device driver or another monitor downstream in the monitor chain must not be blocked. The device driver should not issue the DeRegister device helper call in either of the following situations:

1. It has previously blocked on a monitor chain notification routine call because it cannot process the data received into its monitor chain buffer.
2. The return of a critical data record is pending (for example, a data record is placed into the monitor chain which the device driver is waiting to receive from the monitor chain before processing additional request packets).

Depending on timing and the well-behaved nature of the deregistering monitor, data records may be lost during monitor termination. A monitor that has not stopped taking data from and returning data to the data stream before terminating may inadvertently consume data records.

## Guidelines for a Character Device Driver

For an application to monitor data passing through a character device, the character device driver must provide monitor support. A summary of requirements and restrictions for character device drivers follows:

### Buffer Requirements

The character device driver must include a set of *buffers* in its DATA segments:

1. The character device driver must include in its device header segment, that is, its first data segment (the device driver may have multiple segments), a buffer in which to build the data record that is placed into a monitor chain and sent to its monitors on a MonWrite device helper call.
2. For each data stream that can be monitored, the character device driver must also include in its first data a *monitor chain buffer* that receives filtered data from the monitor chain.

The first WORD of each monitor chain buffer must initially contain the length of the buffer, length WORD inclusive. The length of a monitor chain buffer will define:

1. The *minimum size of all buffers* that are part of the monitor chain

   The length of each of the input and output buffers belonging to all monitors that register with the monitor chain must be greater than or equal to the length of the device driver's monitor chain buffer plus 20 bytes.

2. The *maximum size of a data record placed into the chain* of monitor buffers (that is, the length of the device driver's monitor chain buffer minus 2 bytes).

### Code Requirements

The character device driver must include a set of routines and request handlers in the strategy routine in its CODE segment:

*For each data stream that can be monitored, the character device driver must include in its strategy routine a notification routine that is called by the monitor dispatcher when it has placed a single filtered data record into the device driver's monitor chain buffer.*

The notification routine is called by the monitor dispatcher :

- When the monitor dispatcher automatically moves filtered data from the output buffer of the last monitor in a monitor chain into the device driver's monitor chain buffer
- When the device driver calls the MonWrite device helper to write data into an empty monitor chain.

When the notification routine is called by the monitor dispatcher, the device driver must process the data in its monitor chain buffer before returning to the monitor dispatcher.

*A character device driver must manage its monitor chains: creating them, deleting them, writing data into them, and flushing them.*

For each data stream that can be monitored by a chain of monitors a device driver must define for the monitor dispatcher the location (addresses) of the notification routine and the monitor chain buffer by calling the MonitorCreate device helper routine.

The monitor dispatcher assigns a HANDLE to the chain of monitors for the data stream. The device driver will use this handle when instructing the monitor dispatcher to perform device helper functions on the monitor chain (for example, Register, MonWrite, MonFlush, and DeRegister).

The device driver may call the MonitorCreate device helper routine any time prior to issuing other monitor dispatcher device helper functions:

- At initialization time, that is, when the device driver is installed
- When it receives a monitor open request, if the MonitorCreate device helper routine has not been previously called to define the monitor chain
- When it receives a monitor register IOCTL request, if the MonitorCreate device helper routine has not been previously called to define the monitor chain, and before the Register device helper routine is called to register a monitor with the monitor chain.

A device driver sends data to its monitors by placing it into the monitor chain. When monitors are registered on a monitor chain associated with a device driver's data stream, the device driver must place data received from the device or an application into the monitor chain so that it can be filtered. The device driver builds a data record in its data segment and calls the MonWrite device helper routine to write that record into the monitor chain.

When a monitor chain associated with a device driver's data stream is empty (that is, a monitor chain has been defined during a MonitorCreate device helper call but no monitors have been registered), the device driver may use its monitor support to place a data record directly into its monitor chain buffer by calling the MonWrite device helper routine. The monitor dispatcher automatically calls the device driver's notification routine to process the data.

A device driver may flush all data from the monitor chain associated with a data stream. When a device driver requires that all data has been removed from the monitor chain. it issues a MonFlush device helper call to direct the monitor dispatcher to place a specially marked record, a FLUSH record, into the monitor chain. This record must pass through all monitors in the chain. All monitors in the chain that receive a FLUSH record on a DosMonRead must return it to the monitor chain on a DosMonWrite. No new data records will be placed into the monitor chain until the FLUSH record reaches the device driver's monitor chain buffer.

*A character device driver must handle monitor open, register, and close requests in its strategy routine.*

When an application calls DosMonOpen to get a handle to the device for monitors, an monitor open request is sent to the device driver. In response, a device driver may define a monitor chain for the data stream by calling the MonitorCreate device helper routine, if it has not previously done so.

When an application calls DosMonReg to register a pair of buffers as part of the monitor chain for a specified data stream (see INDEX parameter) for a device, a monitor register request is sent to the device driver. In response, the device driver:

- May call the MonitorCreate device helper routine to define the monitor chain, if it has not previously done so
- Uses the monitor chain handle returned from a previous MonitorCreate device helper call and calls the Register device helper routine to instruct the monitor dispatcher to insert the buffers into the monitor chain for the specified data stream
- Tracks the number of monitors it has registered with each of its monitor chains.

When an application calls DosMonClose to terminate monitoring data passing through a device, a monitor close request is sent to the device driver. In response, the device driver:

- Must, for each monitor chain with which there are monitors currently registered, call the DeRegister device helper routine to instruct the monitor dispatcher to remove all buffers associated with the process that issued the DosMonClose from the monitor chain

  **Note:** A single process may register monitors for more than one data stream for a device. For example, a process may register a keystroke monitor for each OS/2 session. When this process terminates, the keyboard device driver calls the DeRegister device helper for each monitor chain associated with an OS/2 session.

- May, on return from each DeRegister device helper call for a monitor chain, if there are no monitors registered in the monitor chain, call the MonitorCreate device helper with the delete option to delete the monitor chain.

## Special Considerations for Character Device Drivers

This section summarizes conditions that should be considered when writing a character device driver with monitor support.

## Device-specific Monitor Information

The character device driver that provides device monitor support must carefully document the following information for its monitors:

1. The definition of its data streams and their monitor chains.

   The monitor application developer requires this information for the INDEX parameter to the DosMonReg monitor function call.

2. The length of its monitor chain buffers.

   The monitor application developer requires this information so that he can allocate large enough monitor input and output buffers for his monitor.

3. The description of its monitor record format, including a detailed description of the flag usage and expected actions as well as restrictions of data record consumption.

   The monitor application developer requires this information to understand the data flowing through the data stream and any actions or responses that the device driver expects.

4. A description of conditions under which the device driver may block the data stream.

   The monitor application developer must ensure that the device monitor works with and for the device driver, not against it. The monitor application developer must understand expected blockages in the device driver (their causes, their effects) to handle them.

## Performance

The performance of a character device driver with respect to its monitor chains can be affected by several factors.

*Performance can be degraded when interrupts are disabled too long.*

When a monitor chain is empty, a character device driver may place data directly into its monitor chain buffer by calling the MonWrite device helper routine. The monitor dispatcher disables interrupts, writes the data into the monitor chain buffer, calls the device driver's notification routine, and re-enables interrupts when the device driver returns to the monitor dispatcher. If the device driver fails to process the data placed into its monitor chain buffer quickly, interrupts may be disabled too long.

The monitor dispatcher must guarantee that the data placed into the device driver's monitor chain buffer during a MonWrite device helper call is processed before new data can be placed into the buffer on subsequent calls. Because there is no shared semaphore or other means of signalling between the monitor dispatcher and the device driver to protect this data area from being overwritten, the monitor dispatcher protects the data in the buffer by disabling interrupts until the device driver has processed the data.

*Performance can be improved by increasing the size of a monitor chain buffer.*

Since the size of this buffer determines the size of all monitor buffer in a monitor chain, performance will be improved during MonWrite calls to place into the monitor chain, as well as during DosMonRead and DosMonWrite monitor function calls that intercept and return data to the data stream. As the size of the buffers increase, fewer blockages occur.

**Note:** When increasing a *published* size of a device driver's monitor chain buffer, there is a danger of preventing existing monitors from successful monitor registration. Remember, the size of the monitor's input and output buffers is based on the size of the device driver's monitor chain buffer.

# Device Driver Problems

When character device monitors are registered and active, a character device driver may severely and/or permanently impact its data stream or the entire system due to its:

1. Failure to process data received in its monitor chain buffer quickly when the notification routine is called, especially when MonWrite'ing into an *empty* monitor chain.

   Blockages can occur in all monitor buffers in a monitor chain if the device driver is blocked too long.

2. Inability to notify its monitors that there is a problem with the device.

   When a blockage occurs in a device driver because the device is disabled, the device driver can notify its monitors of the problem by placing a special monitor record into the monitor chain. The monitor can inform the user that there is a problem by displaying a message on the screen so that the user can re-enable the device.

   However, blockages in all monitor buffers in the monitor chain due to a blockage at the device driver will prevent the device driver from notifying its monitors of a problem through a special monitor data record. For example, the printer device driver will be unable to send a monitor record to the spooler, if the spooler is blocked.

3. Dependencies on the return of specific types of monitor data records from the monitors to the device driver.

   Documentation on individual device drivers must include requirements for returning special monitor data records to the monitor chain and to the device driver (for example, flush records). Even with well-behaved monitors, however, the device driver cannot guarantee that its monitors will return its critical data. The introduction of special monitor data records into a monitor chain monitored by applications previously written may cause the system to hang.

# Sample Programs

DEMODD.ASM and MONITOR.ASM are sample MASM programs included in the OS/2 Programmer's Toolkit. DEMODD.ASM is a sample character device driver for the DEMOD$ device. MONITOR.ASM is a sample character device monitor for the DEMOD$ device.

## A character device driver (DEMODD.ASM)

DEMODD.ASM demonstrates:

- How a device driver's strategy routine and timer handler are structured
- How queued requests are handled by the timer handler
- How a character device driver provides character device monitor support.

  When a device monitor is registered with the DEMOD$ device, the device driver sends data to the monitor and queues the filtered data returned from the monitor.

The user installs this device driver by including a DEVICE= statement in his CONFIG.SYS file. The user sends characters to the DEMOD$ device and the DEMODD device driver generates speaker tones with pitches corresponding to numeric characters.

The user writes data to the DEMOD$ device by echoing, that is, directing, a string of characters to the DEMOD$ device. The DEMODD device driver strategy routine receives these characters through the request packet interface. If a character device monitor (see MONITOR.ASM) is registered with the monitor chain associated with the DEMODD data stream, all characters are sent to the monitor by calling the MonWrite device helper routine. If no character device monitors are registered, the DEMODD device driver processes the original input characters.

The DEMODD device driver processes characters before queueing them on a character queue. The character string is scanned and only numerics are queued; all other characters are discarded. The timer handler wakes up every few milliseconds and reads the character queue. The timer-interrupt generates speaker tones with a frequency which varies with the numeric character read from the queue.

## A character device monitor (MONITOR.ASM)

MONITOR.ASM demonstrates:

- Structuring a character device monitor
- Registering a monitor with a monitor chain
- Intercepting data from a data stream, filtering that data and returning the filtered data to the data stream

- Handling special monitor records (the flush record)
- Terminating a monitor.

The user runs the MONITOR application program as a detached process. The process registers a monitor with the DEMODD device driver and intercepts all data sent to the DEMOD$ device. The monitor filters the input data to the DEMOD$ device by changing the characters A through J into 0 through 9 respectively. All other characters are unchanged. The monitor returns the filtered data to the data stream. The filtered data alters the function of the device driver: the device driver generates speaker tones for the filtered characters A through J (received as numeric characters 0 through 9).

The monitor terminates itself whenever the character Q is intercepted from the data stream. After the monitor has been terminated, the characters A through J are no longer filtered. The DEMODD device driver generates no speaker tones for these non-numeric characters.

The RUNDEMOD.CMD batch file on the OS/2 sample programs diskette illustrates the function of the DEMODD device driver and how its monitor can alter that function:

1. Characters ABCDEFJ01256579 are sent to the DEMOD$ device that is not monitored by a character device monitor. The DEMODD device driver receives seven non-numeric and eight numeric characters. It generates eight speaker tones corresponding to the numeric characters only.
2. A character device monitor (MONITOR.ASM) is registered with the DEMODD device driver.
3. Characters ABCDEFJ0125679 are sent to the DEMOD$ device. The character device monitor changes the characters A through J to numeric characters 0 through 9. The DEMODD device driver receives seven filtered and eight non-filtered numeric characters. It generates fifteen speaker tones corresponding to the numeric characters.
4. Characters QABCDEFJ0125679 are sent to the DEMOD$ device. When the character device monitor receives the Q, it terminates. The DEMODD device driver receives seven non-numeric and eight numeric characters. Speaker tones are generated for the eight numeric characters only.

# Chapter 9. I/O Privilege Level Code Segments (IOPL)

In a multitasking environment, multiple applications share the same resources, including devices. For example, all applications may send data to the same printer. A single device driver guarantees orderly access to a device by manipulating the device on behalf of all applications sending data to or receiving data from the device. To maintain the integrity of the system, therefore, applications should access the device through the device driver.

In a single-tasking environment, such as DOS, applications do not share resources with other applications. Orderly access to a device is not a problem because a device can be used by only one application at a time. Applications, therefore, can access devices directly or through the device drivers without danger of impacting the system or other applications.

Because OS/2 is a multitasking environment, DOS mode and OS/2 mode applications share the same resources. Applications running in the OS/2 environment that directly access a device, therefore, may impact the system or another application. Because of this, applications in the OS/2 environment should access devices through the device drivers.

There are occasions, however, when it is not appropriate for OS/2 device drivers to provide sufficient I/O service for applications. The overhead (that is, the number of instructions and the time it takes to execute those instructions) required for a device driver to implement a function may be far greater than the overhead required for an application to implement the same function. For example, the device driver implementation of the function of writing a character to a screen would require an IOCtl interface from the application to the device driver and hundreds of instructions in the device driver. An application that can directly access a device, however, can implement this same function with a few instructions and no special interfaces.

In OS/2, Presentation Manager applications require a specialized graphics/video interface not provided by OS/2 device drivers. So that applications or subsystems can provide the additional I/O capability, OS/2 provides a mechanism by which an application or subsystem can directly access a device. An application or subsystem can customize its I/O interface to the device without replacing the device driver. This mechanism is the *I/O privilege level (IOPL) code segment.*

IOPL code segments are special code segments in which an OS/2 mode application or subsystem can:

- Move data to and from a hardware port (through IN/OUT instructions)
- Protect critical sections of code from being interrupted by disabling/ enabling interrupts (through CLI/STI instructions).

  **Note:** Hardware interrupts can be serviced only by device drivers. An application or subsystem cannot respond to or be invoked by a device (hardware) interrupt. An application can only manipulate the state of the interrupts, that is, disable/enable interrupts, in IOPL code segments. Because of this, the IOPL code segment mechanism is primarily used by applications to access non-interrupt driven devices, such as display adapters.

An application can access a device directly through its own IOPL code segments. In this case, the application is *device-dependent.* An application also can access a device indirectly by calling routines in IOPL code segments that are part of a dynamic link library. In this case, the application remains *device-independent*; the dynamic link library is *device-dependent.* For example, OS/2 Presentation Drivers that provide the special graphics/video interface for Presentation Manager applications include IOPL code segments. Presentation Drivers are device-dependent, dynamic link libraries that service device-independent Presentation Manager applications.

## IOPL and the 80286 Protection Model

OS/2 is based on the architecture of the Intel 80286™ processor. Applications and the operating system are protected from each other by a hierarchy of protection, or privilege, levels. The exe-

™ Trademark of the Intel Corporation.

cution of CLI/STI and IN/OUT I/O instructions is restricted to code executing at the I/O privilege level, or a more privileged level. In addition, hardware interrupts can be serviced only at the most privileged level.

Figure 9-1 shows the protection hierarchy provided by the 80286 architecture:

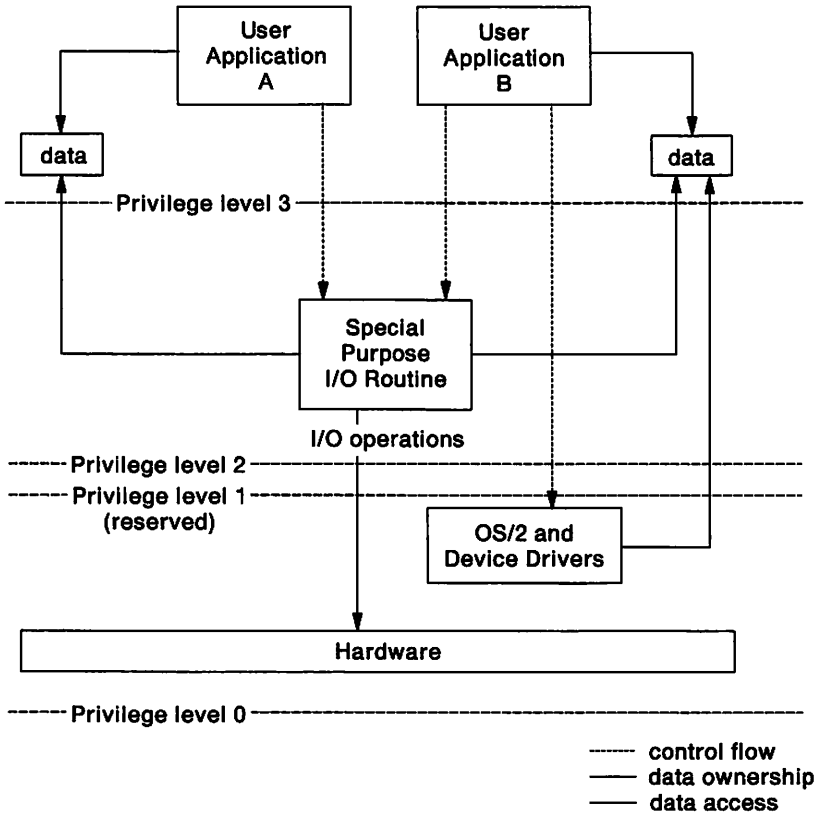



Figure 9-1. Privilege Levels: Hierarchy

OS/2 takes advantage of the hierarchical protection levels imposed by the 80286 architecture. Code executes at the following protection (privilege) levels:

**Privilege level 0:** OS/2 and device drivers
**Privilege level 1:** Reserved
**Privilege level 2:** Special purpose routines (other than device drivers) requiring I/O privilege
**Privilege level 3:** User applications.

At any specified level in the hierarchy, only programs at that level or a more privileged (protected) level can access data at the specified level. For example:

- A user application has access to its own data at privilege level 3 only.
- An I/O module at privilege level 2 has access to both its own data at privilege level 2 and its user application data at privilege level 3.
- OS/2 and device drivers have access to data at all levels.

The 80286 processor does not allow applications to execute the IN/OUT, CLI/STI I/O instructions. Applications run at privilege level 3, and, therefore, cannot normally execute these instructions. However, OS/2 allows applications to execute these instructions from IOPL code segments at privilege level 2 when:

1. The design of the application adheres to the guidelines in the following sections of this chapter.
2. The user provides for it by specifying IOPL=YES in the CONFIG.SYS file.

   The OS/2 IOPL mechanism is a modification of the 80286 protection mechanism. Applications with IOPL may change the system environment. Because of this, OS/2 requires the end user to grant permission for any application to have I/O privilege.

## IOPL versus Device Drivers

Depending on the device's complexity, a non-interrupt-driven device can be controlled by either IOPL code segments or by a device driver. When the device is complex, a device driver is the best method of controlling I/O to the device.

The following table summarizes the characteristics of IOPL code segments and device drivers. This table may help you decide whether a device driver or IOPL code meets your I/O requirements.

| Topic | IOPL Code Segments | Device Drivers |
|---|---|---|
| Interrupts | Cannot receive, register, or process interrupts | Handles interrupts |
| System Services | Can request system services with some limitations | Can request some system services with DevHelp calls |
| Device | Typically used for polled input devices or output devices that are not interrupt-driven | Can be used for any type device |
| Dynamic Linking | Can be called by dynamic linking | Cannot be called by dynamic linking |
| Devices | Best used for less complex devices | Best used for complex devices |
| CONFIG.SYS | Specified in CONFIG.SYS file (IOPL=yes) | Specified in CONFIG.SYS file (Device=*device driver name*) |
| Memory | Occupies memory only when the application using it is called into memory | Remains resident in memory |
| Privilege Level | IOPL available at privilege level 2 | IOPL available at privilege level 0 |
| CLI/STI Instructions | Can issue CLI/STI instructions | Can issue CLI/STI instructions |
| IN/OUT Instructions | Can issue IN/OUT instructions | Can issue IN/OUT instructions |

## System Services and IOPL Code Segments

DOSxxx function calls can be issued from code running at privilege level 2, with the exception of :

**DosReadAsync**
**DosWriteAsync.**

Data items (input or output) associated with all other DOSxxx function calls may be located at privilege level 2, with the exceptions listed below. Applications which violate certain limitations will cause a general protection fault and will be terminated.

**Dynamic Link Initialization** - Dynamic link module initialization is indicated by the keywords INITGLOBAL or INITINSTANCE on the linker LIBRARY statement.

The initialization routine is at the entry point specified on an END statement in one of the source files making up the dynamic link module. The routine specified by either of these keywords must be at privilege level 3. After receiving control, a dynamic link initialization routine may make calls to routines located at privilege level 0, 2, or 3.

If an initialization routine is specified in a segment which has IOPL indicated, any process attempting to use this dynamic link module will cause a general protection fault to occur and will be terminated.

**DosCreateThread** - The initial entry point must be at privilege level 3. If this entry point is in a segment which has IOPL indicated, the DosCreateThread request will cause a general protection fault and the process will be terminated.

**DosExecPgm** - Part of the process of loading a module for DosExecPgm involves loading any dynamic link modules that it references. Processing of a DosExecPgm request cannot cause multiple privilege levels to be invoked. A DosExecPgm request will fail if that module references a dynamic link library which has an initialization routine whose entry point is at privilege level 2.

The initial entry point for the module being loaded must be at privilege level 3. If this entry point is in a segment which has IOPL indicated, a DosExecPgm request for that module will cause a general protection fault and the process will be terminated.

**DosExitList** - The input for this function call is the entry address of a routine which is to be given control when the process terminates. The entry point provided must be at privilege level 3. After receiving control, this routine may make calls to routines located at privilege level 0, 2, or 3.

**DosLoadModule** - Part of the process of loading a dynamic link library is executing any initialization routine it has.

Processing of a DosLoadModule request cannot cause multiple privilege levels to be invoked: A DosLoadModule request will fail if issued at privilege level 2 for a dynamic link library which has an initialization routine. (Initialization routines must execute at privilege level 3.)

If the module's initialization routine is specified in a segment which has IOPL indicated, any process attempting to use this module will cause a general protection fault to occur and that process will be terminated. After receiving control, a dynamic link library initialization routine may make calls to routines located at privilege level 0, 2, or 3.

**DosSendSignal** - If thread 1 of the indicated process is executing at privilege level 2 when this call is issued, then that thread will not be dispatched to handle the signal until after it returns to privilege level 3.

**DosSetVec** - The input for this function call is the entry address of a routine which is to be given control when the specified exception occurs. The entry point provided must be at privilege level 3. After receiving control, the exception handling routine may make calls to routines located at privilege level 0, 2, or 3. If this entry point is in a segment which has IOPL indicated and the indicated exception occurs, the exception will cause a general protection fault to occur and the process will be terminated.

**DosSetSigHandler** - The input for this function call is the entry address of a routine (signal handler) which is to be given control when a signal occurs. The entry point must be at privilege level 3. After receiving control, the signal handler routine may make calls to routines located at privilege level 0, 2, or 3.

If the signal handler address is in a segment which has IOPL indicated and a signal is sent for that handler, a general protection fault will occur and the process will be terminated.

**Dynamic storage management function calls** - The following function calls can be issued at either privilege level 2 or 3, but the target segment cannot be at privilege level 2:

**DosAllocHuge** - The segment is allocated at privilege level 3.
**DosAllocSeg** - The segment is allocated at privilege level 3.
**DosAllocShrSeg** - The segment is allocated at privilege level 3.
**DosGetShrSeg** - The segment being accessed is at privilege level 3.
**DosGetSeg** - The segment being accessed is at privilege level 3.
**DosGiveSeg** - The segment being shared is at privilege level 3.
**DosReallocHuge** - The segment being resized must be at privilege level 3.
**DosReallocSeg** - The segment being resized can be at privilege level 2 or 3.
**DosFreeSeg** - The segment being freed must be at privilege level 3.

**Memory Sub-Allocation function calls** - The sub-allocation segment can be at privilege level 2 or 3 for the following calls:

**DosSubAlloc**
**DosSubFree**
**DosSubSet**

**Message function calls** - The module definition file (.DEF) for the application must contain the statement:

```
'_MSGSEG' CLASS 'MSGSEGCODE' IOPL CONFORMING
```

so that the following function calls can be made from privilege level 2:

**DosGetMessage**
**DosPutMessage**
**DosInsMessage**

See the *OS/2 Version 1.2 Programming Reference* and the *OS/2 Version 1.2 Programming Guide* for a detailed description of the parameters and use of each of these function calls.

## Guidelines for IOPL Code Segment Usage

An application or subsystem should not disable hardware interrupts in an IOPL code segment to protect a critical section of code. The application or subsystem should use semaphores. However, when a critical section of code can be interrupted by a hardware interrupt, an application or subsystem may protect that code by disabling hardware interrupts in an IOPL code segment.

When disabling hardware interrupts, an application or subsystem must do the following:

1. The application or subsystem must first issue DosCLIAccess.
2. The application or subsystem must not reload any segment registers while interrupts are disabled, including doing FAR calls.

   Reloading a segment register can cause a segment-not-present fault that the operating system handles with interrupts enabled. Similarly, if an application were executing in an 80386 paged environment, it would not want to generate a page-not-present exception. Existing 16-bit segmented IOPL application or subsystem code can gaurantee this by ensuring that the instruction pointer (IP) does not cross a multiple of 4K and no data segments other than the stack are referenced for the duration of the CLI/STI sequence. In addition, the stack pointer (SP) must not cross a multiple of 4K for the duration of the CLI/STI sequence.

   A segment register can be reloaded safely without causing a segment fault when the segment has been locked in physical memory by a device driver.
3. The application or subsystem should avoid causing any machine exceptions while executing in an IOPL segment with interrupts disabled.

   The operating system can enable interrupts while handling any machine exception, such as Divide by 0, Overflow, or Invalid Opcode.
4. The application or subsystem must not make system calls with interrupts disabled.

Because interrupts are enabled by the operating system in certain situations, failure of an application or subsystem to adhere to these guidelines can compromise the protection of its data structures or hardware.

# Building an IOPL Code Segment

To create an IOPL code segment in an application or subsystem, you must:

1. Place the privileged I/O instructions (IN/OUT, CLI/STI) into single or multiple code segments.
2. Name the code segments to provide a reference.
3. Identify the segments as IOPL code segments in your module definition file.

4. Request the privilege of disabling/enabling interrupts (using CLI/STI I/O instructions) in an IOPL code segment by calling DosCLIAccess.
5. Request permission to do I/O to ports (that is, port access) in an IOPL code segment by calling DosPortAccess. Identify the port and specify that the call is for gaining access to that port.

   **Note:** When DosPortAccess is called, the privilege of disabling/enabling interrupts is granted automatically. There is no need to make an additional call to DosCLIAccess.
6. Release access to a port by calling DosPortAccess. Identify the port and specify that the call is for releasing access to that port.

   **Note:** Applications which perform I/O to ports in IOPL code segments must release port access from OS/2.
7. Specify IOPL=YES in the CONFIG.SYS file.

## IOPL Sample Program (IOPL2.ASM)

The I/O Privilege Level 2 sample program demonstrates direct I/O to a device from an IOPL code segment. IOPL2.ASM directly accesses the speaker/timer devices. Based on keyboard input from the user, the speaker generates a tone with a frequency equal to the ASCII character code of the keystroke, multiplied by 10.

The module definition file for this sample program illustrates the use of the SEGMENT directive to indicate which code segment has IOPL.

The sample program and its module definition file can be found on the sample programs diskette in the *OS/2 Version 1.2 Programmer's Toolkit.*

# Index


## A

ABIOS eoi placement rule 2-20
ABIOS LID IRQ rule 2-20
ABIOS request block rule 2-20
ABIOSCall, DevHlp 5-7
ABIOSCommonEntry 5-8
ABIOSCommonEntry, DevHlp 5-8
advanced BIOS services
  *See* ABIOSCall, DevHlp
  *See* ABIOSCommonEntry, DevHlp
  *See* FreeLIDEntry, DevHlp
  *See* GetLIDEntry, DevHlp
Advanced BIOS/device driver notes 3-8
AllocGDTSelector, DevHlp 5-9
AllocPhys, DevHlp 5-10
AllocReqPacket, DevHlp 5-11
ANSI and Code pages 6-87
ANSI.SYS 6-83
application I/O 2-3
  *See also* Character Device Monitors
  *See also* IOCtl
  *See also* IOPL
ASYNC control IOCtl commands (category 1)
  break off, 45h 7-10
  break on, 4Bh 7-15
  return COM error, 6DH 7-41
  return COM event information, 72h 7-42
  return COM status, 64h 7-34
  return current baud rate, 61h 7-32
  return DCB parameters, 73h 7-43
  return line characteristics, 62h 7-33
  return modem control output signals, 66h 7-37
  return modem input signals, 67h 7-38
  return number of chars in receive queue, 68h 7-39
  return number of chars in transmit queue, 69h 7-40
  return transmit data status, 65h 7-36
  set baud rate, 41h 7-6
  set dcb parameters, 53h 7-16
  set line characteristics, 42h 7-7
  set modem control signals, 46h 7-11
  start transmit, 48h 7-14
  stop transmit, 47h 7-13
  transmit byte immediate, 44h 7-9
ASYNC Notes 7-18
asynchronous communications/COM 6-1
asynchronous communications/IOCtl summary 7-5
asynchronous communications/SEE COM 7-5
AttachDD, DevHlp 5-12
  *See also* inter-device driver communication
attribute field 2-11
attribute field bits
  clock device 2-12
  device type 2-12
  format 2-12
  Generic IOCtl request 2-12
  Get/Set Logical Device 2-12
  IDC bit 2-12
  NULL 2-12
  removable media 2-12
  shared 2-12
  standard input 2-12
  standard output 2-12

## B

base video handlers
  chaining 6-99
  identification 6-99
  running system installation 6-99
  write support 6-99
bimodal device drivers 2-1
BIOS interrupt rule 2-19
BIOS Parameter Block (BPB) 2-12, 4-5
block device drivers 2-1
block device unit code field 4-1
Block, DevHlp 5-13
boot sector format 4-10
BPB 6-95
BPB, BIOS Parameter Block 2-12, 4-5
BUILD BPB, device driver command 4-9
busy bit 4-2

## C

call Advanced BIOS services, device drivers 3-7
category codes, generic IOCtl 7-1
Category 01 IOCtl Commands 7-5
Category 03 IOCtl Commands 7-46, 7-48, 7-50
Category 04 IOCtl Commands 7-55
Category 05 IOCtl Commands 7-92
Category 07 IOCtl Commands 7-103
Category 08 IOCtl Commands 7-134
Category 09 IOCtl Commands 7-150
Category 10 IOCtl Commands 7-157
Category 11 IOCtl Commands 7-160
character device drivers 2-1
character device monitor IOCtls 7-157
Character Device Monitors
  limitations 8-3
  monitor buffer sizes 8-18
  monitor data, format 8-19
  monitor dispatcher 8-5
  monitor dispatcher device helper 8-31
  monitor function calls 8-11
  monitor mechanism 8-5
  monitor problems and solutions 8-26
  monitor processes 8-6
  monitor support in device drivers 8-31
  monitor termination 8-22
  monitoring data streams 8-2
  performance considerations 8-21
  position in a monitor chain 8-19
  thread priorities 8-20
  well-behaved monitor applications 8-23
character queue management 2-17
  *See also* QueueFlush, DevHlp
  *See also* QueueInit, DevHlp
  *See also* QueueRead, DevHlp
  *See also* QueueWrite, DevHlp
CheckModeProtect 6-29
CheckModeReal 6-29
clear to send (CTS) 6-3
clock device bit 2-12
CLOCKS device driver
  CLOCK device time format 6-63

CLOSE, device driver command 4-15
code page support, printer 6-129
code segments, IOPL 9-1
COM 6-1
command codes
  command code field 4-1
  device driver 4-4
  summary 4-1, 4-4
command-specific field, request header 4-3
commands, IOCtl 7-1
communications device driver 6-1
COM/access authorization 6-21
COM/additional function support 6-2
COM/additional port support 6-2
COM/attachment support 6-2
COM/automatic control 6-6
COM/automatic monitoring 6-6
COM/automatic receive flow control 6-7
COM/automatic receive flow control/XON-XOFF 6-7
COM/automatic transmit flow control 6-7
COM/automatic transmit flow control/XON-XOFF 6-7
COM/AUX 6-16, 6-22
COM/baud rate/return 7-13
COM/baud rate/set 7-5
COM/binary - ASCII 6-18
COM/binary data 6-18
COM/bit rate/return 7-13
COM/bit rate/set 7-5
COM/break 6-7
COM/break replacement character/processing 6-7
COM/break signal/off 7-9
COM/break signal/on 7-14
COM/CLOSE 6-19
COM/CLOSE/automatic receive flow control 6-19
COM/CLOSE/break processing 6-19
COM/CLOSE/DTR 6-19
COM/CLOSE/interrupt level 6-19
COM/CLOSE/last level close 6-19
COM/CLOSE/RTS 6-19
COM/CLOSE/transmit hardware 6-19
COM/CLOSE/transmit immediate processing 6-19
COM/code page support 6-18
COM/CTS 6-3
COM/CTS/return 7-37
COM/custom device support 6-2
COM/data translation 6-18
COM/DCD 6-3
COM/DCD/return 7-37
COM/device driver/close consideration 6-8
COM/device driver/defaults 6-8
COM/device driver/Mode Utility 6-8
COM/device driver/open consideration 6-8
COM/device driver/states 6-8
COM/device names 6-16
COM/device names/Advanced BIOS 6-16
COM/device names/COMn 6-16
COM/device names/determination 6-16
COM/device names/IBM PS/2 6-16
COM/device names/logical ID 6-16
COM/device names/none 6-16
COM/device names/Personal Computer AT 6-16
COM/device names/40: 6-16
COM/DOS mode/considerations 6-22
COM/DOS mode/CTTY 6-22
COM/DOS mode/FAPI 6-22
COM/DOS mode/file system 6-22
COM/DOS mode/IOCtls 6-22
COM/DOS mode/old device drivers 6-22
COM/DOS mode/printing 6-22
COM/DSR 6-3
COM/DSR/return 7-37
COM/DTR 6-3
COM/DTR and RTS/return 7-36
COM/DTR/CLOSE 6-6
COM/DTR/disable 6-6
COM/DTR/enable 6-6
COM/DTR/input handshaking 6-6
COM/DTR/OPEN 6-6
COM/error 6-7
COM/error replacement character/processing 6-7
COM/error/return 7-40
COM/event notification 6-8
COM/event word/clear 7-41
COM/event word/return 7-41
COM/file system 6-19
COM/first level open 6-5
COM/flow control/automatic 6-7
COM/flow control/IOCtls 6-7
COM/flow control/manual 6-7
COM/FLUSH 6-6
COM/initialization 6-16
COM/initialization/CONFIG.SYS ordering 6-16
COM/initialization/DEINSTALL 6-16
COM/initialization/device names 6-16
COM/initialization/DEVICE= 6-16
COM/initialization/filenames 6-16
COM/initialization/IBMPS/2 6-18
COM/initialization/interrupt level 6-17
COM/initialization/logical ID 6-18
COM/initialization/multiple device drivers 6-16
COM/initialization/Personal Computer AT 6-17
COM/initialization/ROM BIOS 40: data area 6-18
COM/initialization/40: area 6-17
COM/input modem control signals 6-3, 6-6
COM/input sensitivity/DSR 6-6
COM/INT 14H 6-22
COM/interrupt driven 6-4
COM/IOCtl 6-4
COM/IOCtl summary 7-5
COM/last level close 6-5
COM/line characteristics 6-7
COM/line characteristics/baud rate 6-7
COM/line characteristics/data bits 6-7, 7-7
COM/line characteristics/parity 6-7, 7-7
COM/line characteristics/return 7-32
COM/line characteristics/set 7-7
COM/line characteristics/stop bits 6-7, 7-7
COM/manual control 6-6
COM/manual monitoring 6-6
COM/marking 6-3
COM/Mode Utility/performance 6-23
COM/modem control input signals/return 7-37
COM/modem control output signals/return 7-36
COM/monitor support 6-18
COM/multiple requests 6-4
COM/multiple requests/ordering 6-5
COM/OFF 6-3
COM/ON 6-3
COM/OPEN 6-19
COM/OPEN/first level open 6-19
COM/OPEN/initialize port 6-19
COM/OPEN/interrupt level 6-19
COM/OPEN/last level close 6-19
COM/OPEN/timer tick 6-19

COM/output handshaking/CTS 6-6
COM/output handshaking/DCD 6-6
COM/output handshaking/DSR 6-6
COM/output modem control signals 6-3
COM/output modem control signals/automatic control 6-6
COM/output modem control signals/manual control 6-6
COM/overlapped input output 6-5
COM/performance 6-23
COM/performance/configuration 6-23
COM/performance/hardware overrun 6-23
COM/performance/receive queue overrun 6-23
COM/Personal Computer AT/adapter 6-2
COM/Personal Computer AT/direct to hardware 6-2
COM/Personal Computer AT/input output addresses 6-2
COM/Personal Computer AT/interrupt levels 6-2
COM/Personal Computer AT/number of ports 6-2
COM/physical interface 6-4
COM/PS/2/adapter 6-2
COM/PS/2/Advanced BIOS 6-2
COM/PS/2/interrupt levels 6-2
COM/PS/2/number of ports 6-2
COM/READ 6-4, 6-19
COM/READ/multiple requests 6-19
COM/READ/ordered requests 6-19
COM/READ/queueing of requests 6-19
COM/READ/receive queue 6-19
COM/READ/receive queue buffer overrun 6-19
COM/READ/time-out processing 6-19
COM/receive queue 6-4
COM/receive queue/return size 7-38
COM/receive queue/return # characters 7-38
COM/RI 6-3
COM/RI/return 7-37
COM/RLSD 6-3
COM/RLSD - DCD 7-5
COM/ROM BIOS 40: data area 6-22
COM/RS232-C 6-3
COM/RTS 6-3
COM/RTS/CLOSE 6-6
COM/RTS/disable 6-6
COM/RTS/enable 6-6
COM/RTS/input handshaking 6-6
COM/RTS/OPEN 6-6
COM/RTS/toggling on transmit 6-6
COM/schedule requests 6-4
COM/spacing 6-3
COM/spooler support/COM to COM 6-18
COM/spooler support/COM to LPT 6-18
COM/spooler support/LPT to COM 6-18
COM/states/automatic receive flow control (XON-XOFF) 6-12
COM/states/automatic transmit flow control (XON-XOFF) 6-12
COM/states/baud rate 6-9
COM/states/break replacement character 6-13
COM/states/break replacement character processing 6-13
COM/states/COM error word 6-11
COM/states/data bits 6-9
COM/states/DTR 6-10
COM/states/DTR Control Mode 6-10
COM/states/DTR Disable 6-10
COM/states/DTR Enable 6-10
COM/states/DTR Input Handshaking 6-10
COM/states/error replacement character 6-13
COM/states/error replacement character processing 6-13
COM/states/event word 6-11
COM/states/input sensitivity using DSR 6-11
COM/states/null stripping 6-13
COM/states/output handshaking using CTS 6-11
COM/states/output handshaking using DCD 6-11
COM/states/output handshaking using DSR 6-11
COM/states/parity 6-9
COM/states/read time-out state 6-14
COM/states/read time-out value 6-14
COM/states/RTS 6-10
COM/states/RTS Control Mode 6-10
COM/states/RTS Disable 6-10
COM/states/RTS Enable 6-10
COM/states/RTS Input Handshaking 6-10
COM/states/RTS toggling on transmit 6-10
COM/states/stop bits 6-9
COM/states/transmit immediate 6-14
COM/states/transmitting break 6-11
COM/states/write time-out state 6-14
COM/states/write time-out value 6-14
COM/states/XOFF character 6-13
COM/states/XON character 6-13
COM/status/return 7-33
COM/system performance/degradation 6-23
COM/timeout processing 6-6
COM/transmit data status/return 7-35
COM/transmit hardware 6-4
COM/transmit queue 6-4
COM/transmit queue/return size 7-39
COM/transmit queue/return # characters 7-39
COM/Tx break status/return 7-32
COM/WRITE 6-4, 6-20
COM/WRITE/multiple requests 6-20
COM/WRITE/ordered requests 6-20
COM/WRITE/queueing of requests 6-20
COM/WRITE/throughput 6-20
COM/WRITE/time-out processing 6-20
COM/WRITE/transmit hardware 6-20
COM/WRITE/transmit queue 6-20
COM/XOFF/simulate 7-12
COM/XON/simulate 7-13
CONFIG.SYS
  device driver button definitions 6-51
  device driver function summary 6-51
  device driver interface requirements 6-51
console device drivers
  device drivers, screen and keyboard 6-64
  keyboard device drivers (KBD$) 6-64
  keyboard initialization 6-65
  keyboard run time operation 6-65
  keystroke monitor data packet 6-66
  keystroke monitors 6-66
contexts, device driver 2-14
control screen cursor 6-83
control sequences 6-83
controlling display mode 6-86
cursor 6-83
cursor backward 6-84
cursor control 6-84
cursor control sequences
  cursor backward 6-84
  cursor down 6-84
  cursor forward 6-84
  cursor position 6-84
  cursor position report 6-84
  cursor up 6-84
  device status report 6-85

cursor control sequences *(continued)*
erase in display 6-85
erase in line 6-85
horizontal position 6-84
keyboard key reassignment 6-87
restore cursor position 6-85
save cursor position 6-85
set graphics rendition 6-86
vertical position 6-84
cursor up 6-84

# D

data carrier detect (DCD) 6-3
data set ready (DSR) 6-3
data terminal ready (DTR) 6-3
DDINSTAL 2-1
*See also* External Device Drivers
DDP 2-1
*See also* External Device Drivers
DEINSTALL 4-21
DEINSTALL hardware interrupts 4-21
DEINSTALL Logical IDs 4-21
DEMODD.ASM 3-14
Deregister 6-48
DeRegister, DevHlp 5-15, 8-35
*See also* Character Device Monitors
DevDone, DevHlp 5-16
DevHlp
*See* function codes
DevHlp character queue structure 2-17
device attribute 2-11
device buffer control IOCtls 7-160
device driver
application access 2-3
application I/O 2-3
attribute 2-11
building 3-13
command codes 4-4
commands 3-4
components 2-13
DEMODD.ASM (sample device driver program) 3-14
design considerations 3-1
device header 2-10
DosDevIOCtl 2-3
hardware interrupt handler 2-14
header 2-10
interrupt handler 3-2
IOCtl commands 3-4
memory management 2-16
monitor support 8-31
monitor support for a character device driver 3-3
OS/2 requests 3-1
performance considerations 3-1
RAM semaphores 2-17
request packet 2-13, 4-1
request packet queue management 2-16
requests to OS/2 3-1
routines 2-13
sample program 3-14
semaphore management 2-17
software interrupt handler 2-14
status word 4-2
step-by-step (building) 3-13
strategy routine 2-13
system semaphores 2-17
timer handler 2-14
device driver *(continued)*
writing 3-13
device driver architecture
bimodal device driver operation 2-15
components of a device driver 2-13
types of device drivers 2-1
device driver commands
BUILD BPB 4-9
FLUSH 4-14
GENERIC IOCtl 4-17
INIT 4-5
LOGICAL DRIVE 4-20
MEDIA CHECK 4-7
NONDESTRUCTIVE READ NO WAIT 4-12
OPEN or CLOSE 4-15
READ 4-11
REMOVABLE MEDIA 4-16
RESET MEDIA 4-19
STATUS 4-13
WRITE 4-11
device driver components 2-13
device driver contexts 2-14
device driver data segment, ABIOS 3-6
device driver DEINSTALL 4-21
device driver examples 3-11
device driver file image 2-10
device driver function calls
DosBeep 2-8
DosCaseMap 2-8
DosChgFilePtr 2-8
DosClose 2-8
DosDelete 2-8
DosDevConfig 2-8
DosDevIOCtl 2-8
DosFindClose 2-8
DosFindFirst 2-8
DosFindNext 2-8
DosGetEnv 2-8
DosGetMessage 2-8
DosOpen 2-8
DosPutMessage 2-8
DosQCurDir 2-8
DosQCurDisk 2-8
DosQFileInfo 2-8
DosQFileMode 2-8
DosRead 2-8
DosWrite 2-8
device driver GET FIXED DISK/LOGICAL UNIT MAP 4-23
device driver header 2-10, 2-11
device driver initialization 2-7
device driver interrupt sharing 2-18
device driver monitor buffers 8-36
*See also* Character Device Monitors
device driver monitor chain buffer 8-32
*See also* Character Device Monitors
device driver monitor code requirements 8-36
*See also* Character Device Monitors
device driver notes/using ABIOS 3-8
device driver notification routine 8-32
*See also* Character Device Monitors
device driver PARTITIONABLE FIXED DISKS 4-22
device driver profile (DDP) 2-1
*See also* External Device Drivers
device driver program model 2-10
device driver sample program 3-14
device drivers
disk device driver 6-89

device drivers, base video handlers 6-99
device drivers, video 6-99
device driver, bimodal operations 2-1
device driver, block 2-1
device driver, character 2-1
device driver, DOS mode 2-6
device driver, dual mode 2-1
Device Driver, EGA.SYS 6-72
device driver, interrupt sharing rules 2-19
device driver, multiple block devices 2-1
device driver, multiple character devices 2-1
device driver, previous-level 2-9
device driver, queueing requests 2-16
device driver, types 2-1
device field, next 2-11
device header 2-10
device header fields
  attribute 2-11
  header 2-11
  name/unit 2-13
  next device header 2-11
  strategy routine 2-12
device helper interface
  ABIOSCall 5-7
  AllocGDTSelector 5-9
  AllocPhys 5-10
  AllocReqPacket 5-11
  AttachDD 5-12
  Block 5-13
  DeRegister 5-15
  DevDone 5-16
  EOI 5-17
  FreeLIDEntry 5-18
  FreePhys 5-19
  FreeReqPacket 5-20
  GetDOSVar 5-21
  GetLIDEntry 5-22
  InternalError 5-23
  Lock 5-24
  MonFlush 5-26
  MonitorCreate 5-27
  MonWrite 5-29
  PhysToGDTSelector 5-30
  PhysToUVirt 5-31
  PhysToVirt 5-33
  ProtToReal 5-35
  PullReqPacket 5-38
  PulParticular 5-37
  PushReqPacket 5-39
  QueueFlush 5-40
  QueueInit 5-41
  QueueRead 5-42
  QueueWrite 5-43
  RealToProt 5-44
  Register 5-46
  RegisterStackUsage 5-47
  ResetTimer 5-48
  ROMCritSection 5-49
  Run 5-50
  SchedClockAddr 5-51
  SemClear 5-53
  SemHandle 5-54
  SemRequest 5-56
  SendEvent 5-57
  SetIRQ 5-58
  SetROMVector 5-59
  SetTimer 5-60
device helper interface *(continued)*
  SortReqPacket 5-61
  TCYield 5-62
  TickCount 5-63
  Unlock 5-64
  UnPhysToVirt 5-65
  UnSetIRQ 5-66
  VerifyAccess 5-67
  VirtToPhys 5-68
  Yield 5-69
device helper services 5-1
device monitor control IOCtl commands (category 10)
  register a monitor, 40h 7-157
device monitors
  *See* Character Device Monitors
device status report 6-85
device support diskette 2-1
  *See also* External Device Drivers
device type bit 2-12
devices, ill-behaved 2-19
devices, well-behaved 2-19
DEVICE=parameter string
  device driver examples 3-11
  device driver header 2-11
  device driver program model 2-10
  driver 4-1
  replacing character device 2-8
disabled draw support 6-26
disk control IOCtls 7-150
disk control IOCtl, get device parameters 7-148
disk control IOCtl, get physical device parameters 7-155
diskette control IOCtl, get device parameters 7-148, 7-155
diskette device driver 6-89
disk, (logical) control IOCtls 7-134
disk, (physical) control IOCtls 7-150
display, primary 6-100
done bit 4-2
DOS extended volume architecture
  BPB and get device parameters 6-95
  creating block devices 6-92
  deleting block devices 6-93
  extended DOS partition architecture 6-91
  installing block devices 6-92
DOS mode device driver 2-6
DOS mode Mouse 6-49
  API (INT 33H) 6-50
  coordinates 6-49
  display modes supported 6-50
  handler/router 6-49
  IOCtl calls 6-49
  motion 6-49
  overview 6-49
  pointer 6-50
DOS mode sharing rule 2-19
DOS mode Software Interrupt Support 3-5
DOS mode support 6-26
DOS mode, interrupt sharing 2-19
DosDevIOCtl 2-3
DosMonClose 8-17
  *See also* Character Device Monitors
DosMonOpen 8-12
  *See also* Character Device Monitors
DosMonRead 8-14
  *See also* Character Device Monitors
DosMonReg 8-12
  *See also* Character Device Monitors

DosMonWrite 8-16
*See also* Character Device Monitors
DrawPointer 6-28
dual mode device drivers 2-1
dynamic link calls, device driver 2-8

# E

EGA.SYS Device Driver 6-72
eoi rule 2-20
EOI, DevHlp 5-17
erase control sequences
erase in display 6-85
erase in line 6-85
erase in display control sequence 6-85
erase in line control sequence 6-85
erasing control sequences 6-85
error bit 4-2
error codes, status word 4-2
events
event mask 6-47
mickeys/row and column 6-47
row and column/mickeys 6-47
time 6-47
examples of device drivers 3-11
Extended DOS Partition 6-91
extended partition 6-90
extended screen and keyboard, using
control sequence syntax 6-83
cursor control sequences 6-84
limitations/restrictions 6-83
extended start-up record 6-91
extended volume 6-91
External Device Drivers
DDINSTAL.EXE 2-1
device driver profile (DDP) 2-2
device support diskette 2-1
installation 2-1

# F

field name 2-13
field, attribute 2-11
fixed disk device driver 6-89
format bit 2-12
FreeLIDEntry, DevHlp 5-18
FreePhys, DevHlp 5-19
FreePointerMemory 6-28
FreeReqPacket, DevHlp 5-20
full draw support 6-26
function codes
character queue management 2-17
function codes 5-1
memory management 2-16
semaphore management 2-17
services and device contexts 5-2
function codes, generic IOCtl 7-1

# G

general device control IOCtl commands (category 11)
flush input buffer, 01h 7-160
flush output buffer, 02h 7-161
query monitor support, 60h 7-163
session switch or termination notification, 41h 7-162
general device control IOCtls 7-160
Generic IOCtl 3-5, 4-17
generic IOCtl commands
*See* ASYNC control IOCtl commands (category 1)
*See* general device control IOCtl commands (category 11)
*See* keyboard control IOCtl commands (category 4)
*See* logical disk IOCtl control commands (category 8)
*See* mouse control IOCtl commands (category 7)
*See* physical disk control IOCtl commands (category 9)
*See* pointer draw control IOCtl commands (category 3)
*See* printer control IOCtl commands (category 5)
*See* screen control IOCtl commands (category 3)
*See* video control IOCtl commands (category 3)
generic IOCtls/COM 7-5
Generic IOCtl, Category 8 6-95
generic IOCtl, category 9 6-96
Get Device Parameters 6-95, 7-148
GET FIXED DISK/LOGICAL UNIT MAP 4-23
GET LOGICAL DRIVE MAP 4-20
GetDOSVar, DevHlp 5-21
GetLIDEntry, DevHlp 5-22
GetPointerMemory 6-30

# H

handlers, base video 6-99
hardware interrupt handler 2-14
hardware interrupt interface, printer 6-140
hardware interrupt management 2-18
hardware interrupt sharing 2-18
hardware interrupts, DEINSTALL 4-21
header, device driver 2-11
horizontal position 6-84

# I

IDC bit 2-12
IDC entry point 2-12
IDC, inter-device driver communication 3-2
Identification, video device handler 6-99
ill-behaved device rule 2-19
ill-behaved devices 2-19
INIT Mode 2-14
INIT, device driver command 4-5
INPUT FLUSH, device driver command 4-14
INPUT STATUS, device driver command 4-13
int return rule 2-20
INT 13H 6-90
INT 2F, screen switch notification 6-82
INT 33H 6-37, 6-49, 6-50
INT 33H - DOS Mode Mouse API
conditional off, function 16h 6-60
get button press information, function 05h 6-54
get button release information, function 6 6-55
get position and button status 6-53
hide pointer 6-53
installed flag and reset, function 0h 6-52
light pen emulation off, function 14h 6-59
light pen emulation on, function 13h 6-59
query save mouse state, function 21h 6-61
read mouse motion counters, function 11h 6-58
restore mouse driver state, function 23h 6-62
save mouse drive state, function 22h 6-62
set dbl speed threshold, function 19h 6-60
set graphic pointer block, function 09h 6-57

INT 33H - DOS Mode Mouse API *(continued)*
set mickey/pixel ratio, function 15h 6-59
set Min & Max horiz position, function 07h 6-56
set Min & Max Vert position, function 08h 6-56
set pointer position, function 04h 6-54
set text pointer, function 10h 6-57
set user-defined subroutine, function 12h 6-58
show pointer 6-52
swap user-defined subroutine, function 20h 6-61
inter-device driver communication 2-12, 3-2
*See also* AttachDD, DevHlp
*See also* IDC bit
InternalError, DevHlp 5-23
interrupt handler, device driver 3-2
interrupt handler, hardware 2-14
interrupt management
*See* EOI, DevHlp
*See* SetIRQ, DevHlp
*See* SetROMVector, DevHlp
*See* UnSetIRQ, DevHlp
Interrupt Mode 2-14
interrupt processing 2-18
interrupt processing, 8259 enabling 2-19
interrupt routine 2-15
interrupt sharing 2-18
interrupt sharing, device dependency 2-18
interrupt sharing, DOS mode interrupt handler 2-19
interrupt sharing, getting an IRQ 2-19
interrupt sharing, ill-behaved device 2-19
interrupt sharing, processing interrupts 2-19
interrupt sharing, restrictions 2-19
interrupt sharing, rules 2-19
interrupt sharing, support of DOS mode I/O 2-19
interrupt sharing, well-behaved device 2-19
interrupt sharing, with BIOS interrupt handlers 2-19
interrupt sharing, 8259 enabling 2-19
interrupt stack, configuring 3-2
*See also* RegisterStackUsage, DevHlp
interrupt 17H interface, printer 6-138
interrupt 21H interface, printer 6-137
Interrupt 33 6-50
interrupt-time 2-14
interrupts, nested 3-2
IOCtl 6-46
character device monitor commands 7-157
command summary 7-1
commands 7-1
general device control 7-160
keyboard control commands 7-55
logical disk control 7-134
mouse control commands 7-103
physical disk control commands 7-150
pointer draw control commands 7-46
printer control commands 7-92
screen control commands 7-50
summary 7-1
video control commands 7-48
IOCtl command summary 7-1
IOCtl commands, Keyboard Control 7-55
IOCtl commands, mouse control 7-103
IOCtl commands, printer control 7-92
IOCtl command, Pointer Draw Control 7-46
IOCtl command, Screen Control 7-50
IOCtl command, Video Control 7-48
IOCtl READ 4-11
IOCTL support of code page and font switching, printer
activate font 6-131
IOCTL support of code page and font switching, printer *(continued)*
query active font 6-131
verify font 6-132
IOCtl WRITE 4-11
IOCtls/COM 7-5
IOCtl, character device monitor 7-157
IOCtl, disk control 7-150
IOCtl, general device control 7-160
IOCtl, logical disk control 7-134
IOCtl, monitor 7-157
IOCtl, physical disk control 7-150
IOMR_GF 6-46
IOMR_GK 6-46
IOMR_GM 6-46
IOMR_GS 6-46
IOMR_MC 6-46
IOMR_NB 6-46
IOMR_QS 6-46
IOMR_RD 6-46
IOMW_DP 6-46
IOMW_EM 6-46
IOMW_GP 6-46
IOMW_RP 6-46
IOMW_SK 6-46
IOMW_SP 6-46
IOMW_SS 6-46
IOPL
characteristics 9-3
IOPL versus device drivers 9-3
privilege levels 9-1
protection model 9-1
irq enforcement rule 2-19
irq mask rule 2-19
irq ownership rule 2-20
I/O services
i/o support for the DOS mode 2-6
i/o support for the OS/2 mode 2-5

# K

KBD 6-87
KbdStringIn 6-87
Kernel Mode 2-14
Keyboard Control 7-55
keyboard control IOCtl commands (category 4)
alter keyboard LEDs, 5AH 7-76
create a logical keyboard, 5DH 7-78
destroy a logical keyboard, 5EH 7-79
get code page ID, 78h 7-87
get input mode, 71h 7-80
get interim character flags, 72h 7-81
get keyboard code page info, 7AH 7-91
get keyboard hardware ID, 7AH 7-90
get keyboard type, 77h 7-86
get session manager hot key 7-85
get shift state, 73h 7-82
Notify change of foreground session, 55h 7-70
peek character data record, 75h 7-84
read character data record, 74h 7-83
set code page ID, 58h 7-74
set code page, 50h 7-56
set input mode, 51h 7-66
set interim character flags, 52h 7-67
set KCB, 57h 7-73
set NLS and custom code page, 5CH 7-77
set read notification, 59h 7-75

keyboard control IOCtl commands (category 4) *(continued)*
set session manager hot key, 56h 7-71
set shift state, 53h 7-68
set typematic rate and delay, 54h 7-69
translate scan code to ASCII, 79h 7-88
keyboard device driver 6-64
keyboard reassignment 6-87
keyboard run time operation 6-65
keystroke monitors 6-66
keys, reassign 6-87

# L

LDT, local descriptor table 2-15
length of request packet field 4-1
linkage field, request header 4-3
Lock, DevHlp 5-24
logical block device 6-91
logical disk control IOCtl commands 7-134
logical disk IOCtl control commands (category 8)
begin format, 04h 7-139
block removable, 20h 7-140
format and verify track, 45h 7-146
get device parameters, 63h 7-148
get logical map, 21h 7-141
lock drive, 00h 7-135
redetermine media, 02h 7-137
set device parameters, 43h 7-142
set logical map, 03h 7-138
unlock drive, 01h 7-136
write/read/verify track, 44h, 64h, 65h 7-144
Logical IDs, DEINSTALL 4-21

# M

master start-up record, partition table 6-94
MEDIA CHECK, device driver command 4-7
memory addressability 2-16
memory management 2-16
*See also* AllocGDTSelector, DevHlp
*See also* AllocPhys, DevHlp
*See also* FreePhys, DevHlp
*See also* Lock, DevHlp
*See also* PhysToGDTSelector, DevHlp
*See also* PhysToUVirt, DevHlp
*See also* PhysToVirt, DevHlp
*See also* Unlock, DevHlp
*See also* UnPhysToVirt, DevHlp
*See also* VerifyAccess, DevHlp
*See also* VirtToPhys, DevHlp
mode 6-86
mode of operation
reset mode (RM) 6-86
set graphics rendition (SGR) 6-86
set mode (SM) 6-86
mode of operation control sequences
set graphics rendition 6-86
modem control signals 7-10
modes, device driver 2-14
MonFlush, DevHlp 5-26, 8-34
*See also* Character Device Monitors
monitor applications, well-behaved 8-23
monitor buffers, sizes 8-18
Monitor Create 6-48
monitor data structures
keystroke 6-66
monitor data, format 8-19
monitor DeRegister 5-15
monitor dispatcher 8-5
monitor dispatcher device helper 8-31
*See also* Character Device Monitors
DeRegister 8-35
MonFlush 8-34
MonitorCreate 8-31
MonWrite 8-33
Register 8-33
Monitor Flush 6-48
monitor function calls 2-5, 8-11
*See also* Character Device Monitors
DosMonClose 8-17
DosMonOpen 8-12
DosMonRead 8-14
DosMonReg 8-12
DosMonWrite 8-16
monitor IOCtls 7-157
monitor management
*See* DeRegister, DevHlp
*See* MonFlush, DevHlp
*See* MonitorCreate, DevHlp
*See* MonWrite, DevHlp
*See* Register, DevHlp
monitor mechanism 8-5
monitor problems and solutions 8-26
monitor processes 8-6
monitor support in a character device driver
*See* Character Device Monitors
monitor support in character device drivers 3-3, 8-7, 8-31
monitor support, limitations 8-3
monitor termination 8-22
monitor thread priorities 8-20
Monitor Write 6-48
MonitorCreate, DevHlp 5-27, 8-31
*See also* Character Device Monitors
monitoring data streams 8-2
monitors
*See* Character Device Monitors
monitors, keystroke 6-66
monitors, mouse 6-48
MonWrite, DevHlp 5-29, 8-33
*See also* Character Device Monitors
MouClose 6-46
MouDeRegister 6-46
MouDrawPtr 6-46, 6-48
MouGetDevStatus 6-46
MouGetEventMask 6-46
MouGetNumButtons 6-46
MouGetNumMickeys 6-46
MouGetNumQueEl 6-46
MouGetPtrShape 6-46
MouGetScaleFact 6-46
MouOpen 6-46
MouReadEventQue 6-46
MouRegister 6-46
MouRemovePtr 6-46
MouRemovePtr. 6-48
Mouse - DOS mode 6-49
Mouse - General Information 6-37
Mouse - OS/2 mode 6-44
Mouse - Screen Resolutions 6-36
Mouse API, OS/2 mode 6-46
Mouse control IOCtl commands 7-103
mouse control IOCtl commands (category 7)

mouse control IOCtl commands (category 7) *(continued)*
assign new mouse event mask, 54h 7-111
display mode change, 51h 7-105
DOS mode address of Pointer Draw Device Driver, 5Bh 7-119
get event queue status, 64h 7-128
get mouse device driver status flags, 62h 7-126
get mouse device's motion sensitivity, 61h 7-125
get mouse event mask, 65h 7-129
get mouse scaling factors, 66h 7-130
get number of buttons supported, 60h 7-124
get pointer screen position, 67h 7-131
get pointer shape, 68h 7-132
impending session switch, 52h 7-109
mark collision area, 58h 7-115
notification of mode switch completion, 5Dh 7-123
OS/2 mode address of Pointer Draw Device Driver, 5AH 7-117
read mouse event queue, 63h 7-127
reassign mouse scaling factors, 53h 7-110
return mouse device driver level/version, 6AH 7-133
session switch completion, 50h 7-104
set mouse device driver status flags, 5CH 7-122
set pointer shape, 56h 7-112
specify pointer position, 59h 7-116
unmark collision area, 57h 7-114
mouse device driver 6-37
coordinates 6-45
device driver packaging 6-37
devices supported 6-37
display modes supported 6-48
DOS mode mouse support 6-49
handler/router 6-45
motion 6-45
mouse installation 6-38
OS/2 mode 6-44
OS/2 mode overview 6-44
overview 6-37, 6-49
pointer draw implementation 6-27
screen resolutions 6-36
Mouse Device Driver Packaging 6-37
Mouse Devices 6-37
Mouse Devices Supported 6-37
Mouse Installation 6-38
Mouse IOCtl 6-46
Mouse IOCtl Calls 6-46
mouse monitors 6-48
Deregister 6-48
MonFlush 6-48
MonitorCreate 6-48
MonWrite 6-48
register 6-48
mouse pointer draw implementation 6-27
CheckModeProtect 6-29
CheckModeReal 6-29
DrawPointer 6-28
executable commands 6-27
FreePointerMemory 6-28
functions supported 6-27
GetPointerMemory 6-30
IDC Interface Data Structures 6-30
pointer draw installation 6-44
RemovePointer 6-28
MouSetDevStatus 6-46
MouSetEventMask 6-46
MouSetPtrShape 6-46, 6-48
MouSetScaleFact 6-46
MouXxx API 6-37, 6-46
multiple character device support per device driver 2-11
multiple device headers per driver 2-1, 2-11

# N

name/units field 2-13
nested interrupts 3-2
*See also* RegisterStackUsage, DevHlp
NONDESTRUCTIVE READ NO WAIT, device driver command 4-12
notes, ASYNC 7-18
NULL bit 2-12

# O

obtaining a Logical ID, device drivers 3-7
OPEN, device driver command 4-15
OS/2 device driver operations 2-15
OS/2 function calls, see function calls also device driver 2-8
OS/2 mode Mouse Support 6-44
API 6-46
coordinates 6-45
display modes supported 6-48
Events 6-47
handler/router 6-45
motion 6-45
MouXxx and IOCtl Calls 6-46
overview 6-44
pointer 6-47
pointer draw installation 6-44
OUTPUT FLUSH, device driver command 4-14
OUTPUT STATUS device driver command 4-13

# P

partition table 6-94
PARTITIONABLE FIXED DISKS 4-22
physical disk control IOCtl commands 7-150
physical disk control IOCtl commands (category 9)
get physical device parameters, 63h 7-155
lock physical drive, 01h 7-151
unlock physical drive, 01h 7-152
write/read/verify track, 44h, 64h, 65h 7-153
PhysToGDTSelector, DevHlp 5-30
PhysToUVirt, DevHlp 5-31
PhysToVirt rule 2-20
PhysToVirt, DevHlp 5-33
Pointer Draw Control 7-46
pointer draw control IOCtl commands (category 3)
get pointer draw address, 72h 7-47
Pointer draw IDC interface data structures 6-30
pointer (screen) device driver 6-26
position rule 2-20
previous-level device drivers 2-9
previous-level device drivers, Init 2-9
previous-level device drivers, Install 2-9
previous-level device drivers, rules 2-9
primary display, identification 6-100
print spooler 6-130
Printer Activate Font IOCtl 7-96
printer control IOCtl commands (category 5)
activate font, 48h 7-96
get frame control, 62h 7-98
get infinite retry, 64h 7-99

printer control IOCtl commands (category 5) *(continued)*
get printer status, 66h 7-100
initialize printer, 46h 7-95
query active font, 69h 7-101
set frame control, 42h 7-93
set infinite retry, 44h 7-94
verify font, 6AH 7-102
printer control IOCtls 7-92
Printer Device Driver
code page support 6-129
hardware interrupt interface 6-140
interrupt 17H interface 6-138
interrupt 21H interface 6-137
IOCTL support of code page and font switching 6-131
monitor dispatcher interface 6-140
position of printer monitors 6-130
Printer IOCTL functions 6-137
printer monitors 6-129
replacing the printer device driver 6-129
request packet interface 6-133
reserved device names 6-129
Timeout Processing 6-132
Printer IOCTL functions
printer monitors
*See* Character Device Monitors
Printer Query Active Font IOCtl 7-101
Printer Verify Font IOCtl 7-102
privilege level code 9-1
process management
*See* Block, DevHlp
*See* DevDone, DevHlp
*See* Run, DevHlp
*See* TCYield, DevHlp
*See* Yield, DevHlp
processing interrupts 2-18
processing interrupts, interrupt sharing 2-19
processor mode services
*See* ProtToReal, DevHlp
*See* RealToProt, DevHlp
ProtToReal, DevHlp 5-35
PullParticular, DevHlp 5-37
PullReqPacket, DevHlp 5-38
PushReqPacket, DevHlp 5-39

## Q

queue linkage field, request header 4-3
QueueFlush, DevHlp 5-40
queueing request packets, device driver 2-16
QueueInit, DevHlp 5-41
QueueRead, DevHlp 5-42
QueueWrite, DevHlp 5-43

## R

RAM semaphores 2-17
READ (input) 4-11
RealToProt, DevHlp 5-44
reassign keys 6-87
Register 6-48
RegisterStackUsage, DevHlp 5-47
*See also* interrupt stack, configuring
*See also* nested interrupts
Register, DevHlp 5-46, 8-33
*See also* Character Device Monitors
removable media bit 2-12
REMOVABLE MEDIA, device driver command 4-16
RemovePointer 6-28
replacing character device drivers 2-8
request handling 2-15
request header 4-1
command-specific data field 4-3
queue linkage field 4-3
status word 4-1
request header command-specific field 4-3
request header queue linkage field 4-3
request header status error codes 4-2
request header status field 4-1
request packet 4-1
request packet format 4-1
request packet header 4-1
request packet queue management 2-16
request queue management
*See* AllocReqPacket, DevHlp
*See* FreeReqPacket, DevHlp
*See* PullParticular, DevHlp
*See* PullReqPacket, DevHlp
*See* PushReqPacket, DevHlp
*See* SortReqPacket, DevHlp
request to send (RTS). 6-3
RESET MEDIA 4-19
ResetTimer, DevHlp 5-48
restore cursor position 6-85
ROMCritSection, DevHlp 5-49
routines, strategy 2-12
RS232-C port/IOCtl summary 7-5
RS232-C/COM 6-1
RS232-C/SEE COM 7-5
rule, ABIOS eoi placement 2-20
rule, ABIOS LID IRQ 2-20
rule, ABIOS request block 2-20
rule, BIOS interrupt 2-19
rule, DOS mode sharing 2-19
rule, eoi 2-20
rule, ill-behaved device 2-19
rule, int return 2-20
rule, irq enforcement 2-19
rule, irq mask 2-19
rule, irq ownership 2-20
rule, PhysToVirt 2-20
rule, position 2-20
rule, search rule 2-20
rule, set irq 2-19
rule, sti entry 2-20
rule, system timer 2-19
Run, DevHlp 5-50

## S

sample character device monitor 8-39
*See also* Character Device Monitors
save cursor position 6-85
SchedClockAddr, DevHlp 5-51
Screen Control 7-50
screen control IOCtl commands (category 3)
abios pass-through, 74H 7-53
allocate a selector with Offset, 75H 7-54
allocate a selector, 70h 7-51
deallocate a selector, 71h 7-52
screen cursor control 6-84
Screen Resolutions 6-36
screen switch notification, DOS mode 6-82

search rule 2-20
semaphore management 2-17
*See also* SemClear, DevHlp
*See also* SemHandle, DevHlp
*See also* SemRequest, DevHlp
SemClear, DevHlp 5-53
SemHandle, DevHlp 5-54
SemRequest, DevHlp 5-56
SendEvent, DevHlp 5-57
serial communications/COM 6-1
serial communications/SEE COM 7-5
serial port/IOCtl summary 7-5
set graphics rendition, control sequence 6-86
set irq rule 2-19
SET LOGICAL DRIVE MAP 4-20
SetIRQ, DevHlp 5-58
SetROMVector, DevHlp 5-59
SetTimer, DevHlp 5-60
shared bit 2-12
shared interrupt architecture 2-18
software interrupt handler 2-14
SortReqPacket, DevHlp 5-61
spooler, print 6-130
standard input bit 2-12
standard output bit 2-12
state of the COM port 6-7
status field 4-1
status field error codes 4-2
status field, request header 4-1
status word 4-1
status word bits
busy 4-2
done 4-2
error 4-2
error code 4-2
sti entry rule 2-20
strategy routine 2-13, 2-15
strategy routines 2-12
support of previous-level device drivers 2-9
system clock management
*See* SchedClockAddr, DevHlp
system semaphores 2-17
system services
*See* GetDOSVar, DevHlp
*See* ROMCritSection, DevHlp
*See* SendEvent, DevHlp
system timer rule 2-19

# T

task-time 2-14
TCYield, DevHlp 5-62
TickCount, DevHlp 5-63
timer handler 2-14
timer services
*See* ResetTimer, DevHlp
*See* SetTimer, DevHlp
*See* TickCount, DevHlp
trusted instructions 9-1

# U

Unlock, DevHlp 5-64
UnPhysToVirt, DevHlp 5-65
UnSetIRQ, DevHlp 5-66
User Mode 2-14
using advanced BIOS 3-6

# V

VerifyAccess, DevHlp 5-67
vertical position 6-84
Video Control 7-48
video control IOCtl commands (category 3)
initialize call vector table, 73h 7-49
video device chaining 6-100
Video Device Drivers
video device drivers, chaining 6-100
video device handler functions
Video Device Handler Interfaces
VirtToPhys, DevHlp 5-68

# W

well-behaved devices 2-19
well-behaved monitor applications 8-23
WRITE (output) 4-11
WRITE (output) WITH VERIFY 4-11
writing input/output privilege level (IOPL) 9-1

# Y

Yield, DevHlp 5-69

# Numerics

8259 Interface, printer 6-140

IBM United Kingdom
International Products Limited
PO Box 41, North Harbour
Portsmouth, PO6 3AU
England

Printed in Denmark by Rosendahl · Esbjerg

